1991 CERN SCHOOL OF COMPUTING

Ystad’s Saltsjöbad Hotel, Ystad, Sweden

23 August – 2 September 1991

PROCEEDINGS

Editor: C. Verkerk
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l’autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s’il s’agit d’utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée. Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s’opposer à toute revendication qu’un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrite dans le présent document.

© Copyright CERN, Genève, 1992

Library and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.

ISSN 0304-2809
ISBN 92-9083-045-X
1991 CERN SCHOOL OF COMPUTING

Ystads Saltsjöbad Hotel, Ystad, Sweden

23 August – 2 September 1991

PROCEEDINGS
Editor: C. Verkerk
ABSTRACT

These Proceedings contain written accounts of the lectures given at the 1991 CERN School of Computing, covering a variety of topics.

Three lecture series introduced the problems to be faced at the future colliders: "Physics at the LHC", "Physics and its impact on the experiments" and "Experiments at future colliders: the first 10 microseconds." Related topics were covered in the lectures: "What have we learned from LEP and HERA?" and "High-level triggers and the case for advanced software engineering in HEP."

"Artificial Neural Networks and HEP" and "Artificial Intelligence in particle physics" were treated in two other lecture series. A special lecture on "The limits of artificial intelligence" is also reproduced.

The field of operating systems received attention in the lectures on "Basics of real-time operating systems" and "A portable, uniform interface to real-time operating systems."

In the area of computer architecture, lectures on "SIMD computers, architecture and algorithms", on "The next generation of transputers and beyond", and on "High-performance RISC, CISC and VLIW computers" were given.

Finally, these Proceedings contain lectures on "Monte Carlo event generation" and "Object Oriented Programming."
Preface

In August 1991 the fourteenth CERN School of Computing was held in Ystad, Sweden. Sixty-nine students from seventeen countries participated.

The School was organised in collaboration with the Particle Physics Section of the Swedish Physical Society. We are extremely grateful to Drs. Leif Jönsson, Torsten Åkesson and Carsten Peterson for the excellent organisation of the School. In particular our most sincere thanks are due to Dr. Leif Jönsson who took the local organisation in his expert hands and deeply impressed everyone with his efficiency and organisational skills.

The School benefitted greatly from the very generous financial contributions from various Organisations and Institutes. The Swedish Research Council and the Royal Academy of Science come first in the list, followed by the Institute of Physics and the Institute of Theoretical Physics of the University of Lund. Our most sincere thanks go to these Institutions; it was their contribution which made the School possible.

The local authorities offered a welcome cocktail and the Chairman of the Representative Committee of Ystad, Mr. Gabrielson honoured the School with his presence. We thank him and the town he represents for their hospitality.

The School was characterised by a very rich lecture programme, which was much appreciated by the participants. The majority of the lectures are reproduced in these Proceedings. We were particularly fortunate to have two special lectures by Professor Roger Penrose, one of which could be included in these Proceedings by courtesy of Blackwell Publishing Company. Professor Penrose and all the other lecturers are sincerely thanked for their efforts and interest in the School.

It has become a tradition for the CERN Schools of Computing to have hands-on exercises for the participants. The School in Ystad was no exception and we are very grateful to Carsten Peterson and Thorsteinn Rögnvaldsson for their thorough preparation and smooth running of the exercises on artificial neural nets. The students greatly appreciated this possibility of getting practical experience with neural nets. Two students from Lund University, Christian Jacobsson and Morten Nyberg, gave assistance during the practical work. We extend our warm thanks to them.

The practical sessions have been made possible through the support of Digital Equipment Corporation and Hewlett-Packard, who provided the workstations necessary for the exercises, free of charge. Messrs. Jan Andersson and Tommy Olsen of DEC and Thomas Malmquist of HP are sincerely thanked for having arranged this. Mr. Huttenberger of NeXT Computer Europe made a NeXT computer available for well attended demonstrations of Object-Oriented Programming techniques.

During the half-day excursion which was organised to visit a few castles and a pre-historic site in the neighbourhood of Ystad, we were very pleased to listen to the expert explanations of our guide, Mrs. Majken Tornblad. Our sincere thanks also to her.

The direction and personnel of the Saltsjöbad Hotel did everything to make our stay a memorable one. We thank particularly Messrs. Peter Weinhandl, Lars Jonsson and Anders Hallin. Special cheers go to our barkeeper!

We express our thanks to Mrs. Ingrid Barnett for her efficient help in organising the School, to the members of the Advisory Committee for putting together an interesting programme and to the Text Processing Service at CERN for producing the Proceedings.

Last but not least, we want to thank our students, whose very active participation turned this School into a valuable experience and who, by telling their colleagues, keep the flame burning!

C. Verkerk
The Editor.
Editor's Note

Professor Penrose's lecture on "The Arrow of Time" is not reproduced in these Proceedings. Chapter 7 of Penrose's book "The Emperor's New Mind" (Oxford University Press, 1989, ISBN 0-19-851973-7) covers the contents of this lecture. The second lecture by Professor Penrose, which was entitled "Artificial Intelligence, the Claims and the Issues", is reproduced by kind permission of Blackwell Publishing Company. It will be included in the book entitled "The Simulation of Human Intelligence" (ed. D. Broadbent), due for publication by Blackwell in October 1992.
ADVISORY COMMITTEE

T. ÅKESSON, Lund University, Sweden
R. BRUN, CERN, Geneva, Switzerland
B.C. CARPENTER, CERN, Geneva, Switzerland
R.F. CHURCHHOUSE, University of Wales, Cardiff, U.K. (Chairman)
R.W. DOBINSON, CERN, Geneva, Switzerland
L. JOENSSON, Lund University, Sweden
C. PETERSON, Lund University, Sweden
J.J. THRESHER, CERN, Geneva, Switzerland
A. VAN DAM, Brown University, Providence, U.S.A. (Corresponding Member)
C. VERKERK, CERN, Geneva, Switzerland (Scientific Secretary)
D.O. WILLIAMS, CERN, Geneva, Switzerland
I. BARNETT, CERN, Geneva, Switzerland (School Secretary)
# CONTENTS

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>V</td>
</tr>
<tr>
<td>Editor's Note</td>
<td>VI</td>
</tr>
<tr>
<td>Advisory Committee</td>
<td>VII</td>
</tr>
<tr>
<td><strong>R. Penrose</strong></td>
<td></td>
</tr>
<tr>
<td>Artificial Intelligence. Setting the Scene: the Claim and the Issues</td>
<td>1</td>
</tr>
<tr>
<td><strong>G. Flügge</strong></td>
<td></td>
</tr>
<tr>
<td>Physics at the Large Hadron Collider LHC</td>
<td>23</td>
</tr>
<tr>
<td><strong>C. N. P. Gee</strong></td>
<td></td>
</tr>
<tr>
<td>What Have We Learned from LEP and HERA?</td>
<td>51</td>
</tr>
<tr>
<td><strong>J. Renner Hansen</strong></td>
<td></td>
</tr>
<tr>
<td>The Physics at the LHC and its Impact on the Experiments</td>
<td>63</td>
</tr>
<tr>
<td><strong>L. Mapelli</strong></td>
<td></td>
</tr>
<tr>
<td>Collisions at Future Supercolliders: the First 10 Microseconds</td>
<td>79</td>
</tr>
<tr>
<td><strong>C. Peterson and T. Rögnvaldsson</strong></td>
<td></td>
</tr>
<tr>
<td>An Introduction to Artificial Neural Networks</td>
<td>113</td>
</tr>
<tr>
<td><strong>K.-H. Becks and A. Hemker</strong></td>
<td></td>
</tr>
<tr>
<td>An Artificial Intelligence Approach to Data Analysis</td>
<td>171</td>
</tr>
<tr>
<td><strong>T. Nash</strong></td>
<td></td>
</tr>
<tr>
<td>High Energy Experiment Triggers and the Thrustworthiness of Software</td>
<td>180</td>
</tr>
<tr>
<td><strong>D. Parkinson</strong></td>
<td></td>
</tr>
<tr>
<td>SIMD Processor Arrays</td>
<td>204</td>
</tr>
<tr>
<td>SIMD Algorithms</td>
<td>212</td>
</tr>
<tr>
<td>Factorizing Large Integers</td>
<td>221</td>
</tr>
<tr>
<td><strong>T. Sjöstrand</strong></td>
<td></td>
</tr>
<tr>
<td>Monte Carlo Event Generation for LHC</td>
<td>227</td>
</tr>
<tr>
<td><strong>D. Kemp</strong></td>
<td></td>
</tr>
<tr>
<td>Basics of Real-time Operating Systems</td>
<td>250</td>
</tr>
<tr>
<td><strong>P. F. Kunz</strong></td>
<td></td>
</tr>
<tr>
<td>Object Oriented Programming</td>
<td>271</td>
</tr>
</tbody>
</table>
R. Shepherd, D. May and P. Thomson
Next Generation Transputers and Beyond–1
Transputers and Routers:
Components for Concurrent Machines 295

R. Shepherd
Next Generation Transputers and Beyond–2
T9000: Superscalar Transputer 309

R. Shepherd, D. May and C. Barnaby
Next Generation Transputers and Beyond–3
Towards General Purpose Parallel Computers 323

R. D. Russel
A Portable Uniform Interface to Real-time Operating Systems 330

G. S. Rao
High Performance CISC, RISC and VLIW Processors 400

List of participants 402
1991 CERN SCHOOL OF COMPUTING

Ystads Saltsjöbad Hotel, Ystad, Sweden
23 August – 2 September 1991

PROCEEDINGS
Editor: C. Verkerk
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.
Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les institutions de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

© Copyright CERN, Genève, 1992

LibRARY AND SCIENTIFIC COPYRIGHTS RESERVED IN ALL COUNTRIES OF THE WORLD. THIS REPORT, OR ANY PART OF IT, MAY NOT BE REPRODUCED OR TRANSLATED WITHOUT WRITTEN PERMISSION OF THE COPYRIGHT HOLDER, THE DIRECTOR-GENERAL OF CERN. HOWEVER, PERMISSION WILL BE FREELY GRANTED FOR APPROPRIATE NON-COMMERCIAL USE.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.

ISSN 0304-2809
ISBN 92-9083-045-X
1991 CERN SCHOOL OF COMPUTING

Ystads Saltsjöbad Hotel, Ystad, Sweden
23 August – 2 September 1991

PROCEEDINGS
Editor: C. Verkerk
ABSTRACT

These Proceedings contain written accounts of the lectures given at the 1991 CERN School of Computing, covering a variety of topics.

Three lecture series introduced the problems to be faced at the future colliders: "Physics at the LHC", "Physics and its impact on the experiments" and "Experiments at future colliders: the first 10 microseconds." Related topics were covered in the lectures: "What have we learned from LEP and HERA?" and "High-level triggers and the case for advanced software engineering in HEP."

"Artificial Neural Networks and HEP" and "Artificial Intelligence in particle physics" were treated in two other lecture series. A special lecture on "The limits of artificial intelligence" is also reproduced.

The field of operating systems received attention in the lectures on "Basics of real-time operating systems" and "A portable, uniform interface to real-time operating systems."

In the area of computer architecture, lectures on "SIMD computers, architecture and algorithms", on "The next generation of transputers and beyond", and on "High-performance RISC, CISC and VLIW computers" were given.

Finally, these Proceedings contain lectures on "Monte Carlo event generation" and "Object Oriented Programming."
Preface

In August 1991 the fourteenth CERN School of Computing was held in Ystad, Sweden. Sixty-nine students from seventeen countries participated.

The School was organised in collaboration with the Particle Physics Section of the Swedish Physical Society. We are extremely grateful to Drs. Leif Jönsson, Torsten Åkesson and Carsten Peterson for the excellent organisation of the School. In particular our most sincere thanks are due to Dr. Leif Jönsson who took the local organisation in his expert hands and deeply impressed everyone with his efficiency and organisational skills.

The School benefited greatly from the very generous financial contributions from various Organisations and Institutes. The Swedish Research Council and the Royal Academy of Science come first in the list, followed by the Institute of Physics and the Institute of Theoretical Physics of the University of Lund. Our most sincere thanks go to these Institutions; it was their contribution which made the School possible.

The local authorities offered a welcome cocktail and the Chairman of the Representative Committee of Ystad, Mr. Gabrielson honoured the School with his presence. We thank him and the town he represents for their hospitality.

The School was characterised by a very rich lecture programme, which was much appreciated by the participants. The majority of the lectures are reproduced in these Proceedings. We were particularly fortunate to have two special lectures by Professor Roger Penrose, one of which could be included in these Proceedings by courtesy of Blackwell Publishing Company. Professor Penrose and all the other lecturers are sincerely thanked for their efforts and interest in the School.

It has become a tradition for the CERN Schools of Computing to have hands-on exercises for the participants. The School in Ystad was no exception and we are very grateful to Carsten Peterson and Thorsteinn Rögnvaldsson for their thorough preparation and smooth running of the exercises on artificial neural nets. The students greatly appreciated this possibility of getting practical experience with neural nets. Two students from Lund University, Christian Jacobsson and Morten Nyberg, gave assistance during the practical work. We extend our warm thanks to them.

The practical sessions have been made possible through the support of Digital Equipment Corporation and Hewlett-Packard, who provided the workstations necessary for the exercises, free of charge. Messrs. Jan Andersson and Tommy Olsen of DEC and Thomas Malmqvist of HP are sincerely thanked for having arranged this. Mr. Huttenberger of NeXT Computer Europe made a NeXT computer available for well attended demonstrations of Object-Oriented Programming techniques.

During the half-day excursion which was organised to visit a few castles and a pre-historic site in the neighbourhood of Ystad, we were very pleased to listen to the expert explanations of our guide, Mrs. Majken Tornblad. Our sincere thanks also to her.

The direction and personnel of the Saltsjöbad Hotel did everything to make our stay a memorable one. We thank particularly Messr. Peter Weinhandl, Lars Jonsson and Anders Hallin. Special cheers go to our barkeeper!

We express our thanks to Mrs. Ingrid Barnett for her efficient help in organising the School, to the members of the Advisory Committee for putting together an interesting programme and to the Text Processing Service at CERN for producing the Proceedings.

Last but not least, we want to thank our students, whose very active participation turned this School into a valuable experience and who, by telling their colleagues, keep the flame burning!

C. Verkerk
The Editor.
Editor's Note

Professor Penrose's lecture on "The Arrow of Time" is not reproduced in these Proceedings. Chapter 7 of Penrose's book "The Emperor's New Mind" (Oxford University Press, 1989, ISBN 0-19-851973-7) covers the contents of this lecture. The second lecture by Professor Penrose, which was entitled "Artificial Intelligence, the Claims and the Issues", is reproduced by kind permission of Blackwell Publishing Company. It will be included in the book entitled "The Simulation of Human Intelligence" (ed. D. Broadbent), due for publication by Blackwell in October 1992.
ADVISORY COMMITTEE

T. ÅKESSON, Lund University, Sweden
R. BRUN, CERN, Geneva, Switzerland
B.C. CARPENTER, CERN, Geneva, Switzerland
R.F. CHURCHHOUSE, University of Wales, Cardiff, U.K. (Chairman)
R.W. DOBINSON, CERN, Geneva, Switzerland
L. JOENSSON, Lund University, Sweden
C. PETERSON, Lund University, Sweden
J.J. THRESHHER, CERN, Geneva, Switzerland
A. VAN DAM, Brown University, Providence, U.S.A. (Corresponding Member)
C. VERKERK, CERN, Geneva, Switzerland (Scientific Secretary)
D.O. WILLIAMS, CERN, Geneva, Switzerland
I. BARNETT, CERN, Geneva, Switzerland (School Secretary)
CONTENTS

Preface V
Editor's Note VI
Advisory Committee VII

R. Penrose 1
Artificial Intelligence. Setting the Scene: the Claim and the Issues

G. Flügge 23
Physics at the Large Hadron Collider LHC

C. N. P. Gee 51
What Have We Learned from LEP and HERA?

J. Renner Hansen 63
The Physics at the LHC and its Impact on the Experiments

L. Mapelli 79
Collisions at Future Supercolliders: the First 10 Microseconds

C. Peterson and T. Rögnvaldsson 113
An Introduction to Artificial Neural Networks

K.-H. Becks and A. Hemker 171
An Artificial Intelligence Approach to Data Analysis

T. Nash 180
High Energy Experiment Triggers and the Thrustworthiness of Software

D. Parkinson 204
SIMD Processor Arrays
SIMD Algorithms
Factorizing Large Integers

T. Sjöstrand 227
Monte Carlo Event Generation for LHC

D. Kemp 250
Basics of Real-time Operating Systems

P. F. Kunz 271
Object Oriented Programming
R. Shepherd, D. May and P. Thomson
Next Generation Transputers and Beyond–1
Transputers and Routers:
Components for Concurrent Machines 295

R. Shepherd
Next Generation Transputers and Beyond–2
T9000: Superscalar Transputer 309

R. Shepherd, D. May and C. Barnaby
Next Generation Transputers and Beyond–3
Towards General Purpose Parallel Computers 323

R. D. Russel
A Portable Uniform Interface to Real-time Operating Systems 330

G. S. Rao
High Performance CISC, RISC and VLIW Processors 400

List of participants 402
Artificial Intelligence.
Setting the Scene: the Claim and the Issues

by Roger Penrose

Mathematical Institute, Oxford

Opening lecture†, Wolfson Lecture series on Artificial Intelligence;

Artificial minds?

The advent of high-speed electronic computer technology has provided a new reality, and perhaps also even an urgency, to the arguments concerning some of the most basic and long-standing of philosophical issues. If computers can, before too long, achieve a genuine artificial intelligence, then may they not also have artificial minds? We are used to the idea of artificial hips, or legs or arms, or artificial kidneys or even hearts. Why not an artificial brain? Would an artificial brain have to have a mind? Somehow, we feel that it would not be much use to us if it did not!

The whole question hangs on the resolution of deep philosophical issues. What is a mind? Are minds merely features that inevitably arise in association with the physical actions of (say human) brains? Is the essential feature of such a physical action the carrying out of an immensely complicated computation, of the particular sort that one supposes is being performed by a human brain during the act of thinking? Or is there something else that the brain does that cannot be described in merely computational terms? Perhaps minds are not even things that can be considered in scientific terms at all. Perhaps minds do not really exist.

Electronic computers offer us the possibility of objects that might ultimately be able to calculate more effectively in every way than biological brains, being not limited by biological restrictions such as imprecise structuring and the need to grow from a single cell. Already computers can be programmed to play chess better than all but a very small number of human experts. It seems not unreasonable to suppose that in many other areas where human intelligence had seemed to reign supreme the computers will overtake even the best of us. Are such "artificially intelligent" systems really intelligent? Does actual intelligence require a degree of awareness and understanding of what one is doing?

In my own use of the word "intelligence", I would take this last to be the case. I would not use the word "intelligence" unless some actual understanding

---

* This account is partially based on a draft of part of the opening chapter of a new book by the author, to be published, in due course, by the Oxford University Press.
† Reproduced by kind permission of Blackwell Publishing Company, Oxford.
is involved; nor would I like to use the word "understanding" unless I am prepared to believe that there is some element of awareness present. This does not remove the problem, of course. How will we be able to tell if an electronic system is "actually" intelligent, rather than merely simulating intelligence, if we need to know whether it is aware or not in order to be able to tell the difference? How might we be able to tell if an electronic system is aware or not? Indeed, how can we tell whether human beings other than ourselves are aware or not? With human beings, we try to judge this from the way they talk or move, by their actions, by their writings or other creations, and so on — all external characteristics of their behaviour. Should we not apply similar operational criteria to electronically controlled robots? Would such a robot that appears to behave intelligently necessarily "actually" possess awareness and understanding? Would it be possible to tell? Does the question have genuine meaning? While I shall certainly not be able to provide definitive answers to these age-old canundrums, I shall outline, below, various possible alternative viewpoints; and I shall also give reasons for the beliefs on this issue that I hold to myself.

The prospect of genuine artificial intelligence

While, on the whole, rather little has been achieved by artificial intelligence to date — in any important area of intellectual expertise where genuine human understanding and insight has seemed to be important — there have been some very impressive developments in recent years, most particularly the development of powerful chess computers. But for the most part, the very optimistic claims that had sometimes been made by proponents of artificial intelligence and promoters of expert systems, etc., have not yet been fulfilled. But these are still very early days, if we are to consider what artificial intelligence might ultimately achieve. There would appear to be some enormous potential advantages over brains that electronic computers could eventually make use of. Electronic circuits are already about a million times faster that the firing of the neurons in the brain. Furthermore, they have an immense precision in timing and accuracy of action that is in no way shared by our own neurons. There is a great deal of randomness in the brain's "wiring" that could apparently be vastly improved upon by the deliberate and precise organization of electronic printed circuits. Moreover, in those places where the brain now does have an advantage, it would seem that this advantage may be short lived. In its total neuron number — some hundreds of thousands of millions — the brain is perhaps temporarily ahead (although this depends, to some extent, upon how one does that calculation, for example because different computer units can always be added together to form larger and larger ones). Also, there are, on the average, a good deal more connections between different neurons than there are connections between transistors in a computer. But it is clear, especially when one takes into account the rapid increase in computer technology over the years, that these few remaining numerical advantages that the brain may have will not last longer than a few decades at the very most. Moreover, there may well be technological revolutions waiting in the wings, such as the replacing of the wires and transistors of our present computers by appropriate optical (laser) devices, thereby achieving enormous increases in speed, power, and minituarization. There seems to be little doubt that on any issue of merely computing power, if computers do not have the advantage over brains already, then they will certainly have it before too long.
But are the relevant issues merely those of computing power, or speed, or memory, or perhaps of the detailed way in which things are "wired up"? Might we, on the other hand, be doing something with our brains that cannot be described in computational terms at all? How do our feelings of conscious awareness — of happiness, pain, love, aesthetic sensibility, will, understanding, etc. — fit into such a computational picture? Will the computers of the future indeed actually have minds? Does it make sense to talk about such things in scientific terms at all; or is science in no way competent to address issues that relate to the human mind? It seems to me that there are at least four different viewpoints (or extremes of viewpoint) that one may reasonably hold on the matter:

A. All thinking is computation, and the mere carrying out of appropriate computations will evoke feelings of conscious awareness.

B. It is the brain's physical action that evokes awareness; and any physical action can in principle be simulated computationally. But computational simulation by itself cannot evoke awareness.

C. Appropriate physical action of the brain evokes awareness, but this physical action cannot even be properly simulated computationally.

D. Awareness cannot be explained in physical, computational, or any other scientific terms.

The point of view expressed in D, which negates the physicalist position altogether and regards the mind as something that is entirely inexplicable in scientific terms, is the viewpoint of the mystic; and at least some ingredient of D seems to be involved in the acceptance of religious doctrine. My own position is that questions of mind, though they lie uncomfortably with present-day scientific understanding, should not be regarded as being forever outside the realms of science. If science is as yet incapable of saying much that is of significance concerning matters of the mind, then science itself will have to change. While I reject mysticism in its negation of scientific criteria for the furtherance of knowledge, I believe that within an expanded science and mathematics there will be found sufficient scope to accommodate even what is needed for an understanding of mind. I shall expand on some these ideas a little bit later on, but for the moment it will be sufficient to say that under my own viewpoint, I am rejecting D; and I am attempting to move forward along the path that science has set out for us. I believe that this route will eventually reveal enough mysteries of its own, so that it may not seem so strange if the mystery of mind may be ultimately explicable in terms of it.

Let us consider what seems to be the opposite extreme to D: the viewpoint A. This is what referred to as strong AI (strong artificial intelligence) or functionalism. It is regarded by some as the only viewpoint that an entirely scientific attitude allows. Others would take A to be an an absurdity that is barely worth serious attention. It is the viewpoint that the arguments of my book, The Emperor's New Mind (Penrose 1988) were most specifically directed against. The length of that book alone (466 pages in hardback) should make it clear that, while I do not myself believe that A is correct, I do regard it as a serious possibility that is worthy of considerable attention. It is an implication of a highly operational attitude to science, where, also, the physical world is taken to operate entirely
computationally. In one extreme of this view, the universe itself is taken to be, in effect, a gigantic computer, and appropriate sub-calculations that this computer performs will evoke the feelings of awareness that constitute our conscious minds. I suppose that this viewpoint — that physical systems are to be regarded as merely computational entities — stems partly from the belief that physical objects are themselves merely "patterns of information", in some sense.

Even if we do not think that it is appropriate to regard the universe as simply being a computer, we may feel ourselves operationally driven to viewpoint $A$. Suppose that we have a robot that is controlled by a computer and which responds to questioning exactly as a human would. We ask it how it feels, and find that it answers in a way that is entirely consistent with its actually possessing feelings. It tells us that it is aware, that it is happy or sad, that it can perceive the colour red, and that it worries about questions of "mind" and "self". It may even give expression to a puzzlement about whether or not it should accept that other beings (especially human beings) are to be regarded as possessing a consciousness similar to the one that it feels itself. Why should we disbelieve its mere claims to be aware, to wonder, to be joyful, or to feel pain, when it might seem that we have as little to go on with respect to other human beings whom we do accept as being conscious? The operational argument does, it seems to me, have some considerable force, even if it is not entirely conclusive. If all the external manifestations of a conscious brain can indeed be simulated entirely computationally, then there would indeed be a case for accepting that its internal manifestations — consciousness itself — are also present in association with such a simulation.

What about $B$? I think that it is the viewpoint that many would regard as the "common sense" one. Like $A$, it affirms a view that all the physical objects of this world must behave according to a science that, in principle, allows that they can be computationally simulated. On the other hand, it strongly denies the operational claim that a thing that behaves externally as a conscious being would must necessarily be conscious itself. As the philosopher John Searle has stressed, a computational simulation of a physical process is indeed different from that process itself. (A computer simulation of a hurricane, for example, is certainly no hurricane!) On view $B$, the presence or absence of consciousness would depend very much on what actual physical object is carrying out a computation. Thus, the action of a biological brain might evoke consciousness, while its electronic simulation might not. It is not necessary, in viewpoint $B$, for this distinction to be between biology and physics. But the actual physical constitution of the object in question (say a brain), and not just its computational action, is regarded as all-important.

The viewpoint $C$ is the one which I believe myself to be closest to the truth. It is more of an operational viewpoint than $B$ since it asserts that there are external manifestations of conscious objects (say brains) that differ from the external manifestations of a computer: consciousness cannot even be properly simulated computationally. I shall be giving my reasons for this belief in due course. Since $C$, like $B$, maintains the physicalist standpoint that minds arise as manifestations of the behaviour of certain physical objects (brains — although not necessarily only brains), it follows that an implication of $C$ is that not all physical action can be properly simulated computationally. It is not completely clear whether present-day physics allows for the possibility of an action that is in
principle impossible to simulate on a computer. Rather little is known of a precise mathematical nature on this issue. My own opinion is that such non-computational action would have to be found in an area of physics that lies outside the presently-known physical laws. Later on, I shall briefly indicate some of the reasons, coming from within physics itself, for believing that a new understanding is indeed needed, in an area that lies intermediate between the "quantum level" of molecules, atoms and subatomic particles and the "classical level" of everyday objects. However, it is not by any means universally accepted, among physicists, that such a new physical theory is required.

What of the future?

What do the viewpoints $A, B, C, D$ tell us to expect for the future of this planet? According to $A$, there will come a stage when appropriately programmed supercomputers will reach — and then race beyond — all human mental capabilities. Of course different people who hold to $A$ could have very different views as to the time-scales involved in this. Some might reasonably take the line that it will be many centuries before computers will reach our level, so little being presently understood about the calculations that the brain must indeed be performing (they would claim) in order to achieve the subtlety of action that we undoubtedly attain — a subtlety that would be necessary before appreciable "awareness" would take place. Others argue for a much shorter time-scale. In particular, Hans Moravec, in his book *Mind Children* (1988), makes a reasoned case in support of his claim that "human equivalence" will be reached in a mere forty years, basing his arguments on the rate at which computer technology has moved forward at an accelerating rate over the past half century, and on the proportion of the brain's activity that has already been successfully simulated. (Some have argued for a much shorter time-scale, sometimes, even, where the predicted date for human equivalence has already passed!) Lest the reader feel dismayed by the prospect of being overtaken by computers in (say) forty years' time, hope is offered — indeed promised — by the assured prospect of our being able to transfer our "mind programs" into the shining metallic bodies of the robots of our choice, thereby obtaining for ourselves a form of immortality (Moravec 1988).

Such optimism is not available to the holders of viewpoint $B$, however. Their standpoint does not differ from $A$ with regard to what computers will ultimately be able to achieve. Whether it would take centuries or a mere forty years, computers would again be expected to reach human equivalence, and then race beyond whatever we are able to achieve with our relatively puny brains. Now the option is not open to us to "join" the computer-controlled robots, and it would seem that we must resign ourselves to the prospect of a planet ultimately controlled by sentient machines! Of all the viewpoints $A, B, C, D$, it seems to me that it is $B$ that offers the most pessimistic view of the future of our planet — despite its apparently "common-sense" nature!

According to $C$ or $D$, on the other hand, it would be expected that the computers, as we understand them today, would (or should) always remain subservient to us, no matter how far they advance with respect to speed, capacity, and logical design. Viewpoint $C$, however, is open with regard to future scientific developments that might lead to the construction of devices — not based on
today's concept of computers — which could achieve actual intelligence and awareness. Since we at present lack almost all the scientific understanding that would be necessary, let alone any of the technological knowhow, little can be gained from such speculation at the present time.

Irrespective of such digressions into futurology, we must ask: what is it that we actually do with our brains? Is "computing", in one form or another, the only possibility? Might we, on the other hand, sometimes be doing something else with our brains and minds — something very different from computation, for which our very awareness and (apparent?) free will may be playing an essential role? Are these things that can be discussed at all in anything like scientific terms — as would be denied by viewpoint D? I believe that they can be, and that viewpoint C is a genuine scientific possibility, although we must be prepared for the eventuality that our scientific criteria and methods may undergo subtle but important shifts. We must be prepared to examine clues that may present themselves in unexpected ways, in areas of genuine understanding that may at first appear to be largely irrelevant.

The Non-computability of Mathematical Thinking

In a moment, we shall need to turn to some mathematics. It is there that our thinking processes have their purest form. If thinking is just carrying out a computation of some kind, then we ought to be able to see this most clearly in our mathematical thinking. Yet, remarkably, the very reverse turns out to be the case. It is within mathematics that we find the clearest evidence that there must actually be something in our conscious thought processes that eludes computation.

Nevertheless, one might reasonably ask why we shall need to turn to mathematics at all, in order to find a suitable area in which to demonstrate that there must be something non-computational in conscious thought processes. Indeed, mathematics is certainly very far from being the only animal activity requiring consciousness. It is a very specialized human activity, whereas the phenomenon of consciousness is commonplace, being likely to be present in much non-human as well as human mental activity. The reason for needing to address the question of consciousness here in a mathematical context is that it is only within mathematics that we can expect to find anything approaching a rigorous demonstration that some, at least, of conscious activity must be non-computational in nature. (The issue of computation, by its very nature, is after all, a mathematical one.) If it can be demonstrated that whatever we do with our brains or minds when we consciously understand mathematics is different from anything that we can achieve by use of a computer, then the possibility of non-computational thinking generally has been established. Moreover, since understanding seems to be a conscious activity, it is conscious thinking that seems to have this non-computational character.

Of course consciousness has many other manifestations. The perception of the colour red, for example, is something that requires consciousness, as is the sensation of pain, the appreciation of a melody, the bringing to mind of an early memory, the willed action to get up from one's bed, and so on. Some might dispute an all-embracing role of some single concept of consciousness in all of these various manifestations. In my own view, there is indeed a unified concept
of "consciousness" that is central to all these separate aspects of mentality, but it will not be necessary to worry about this here. Let us just concentrate on "understanding", and this will include "mathematical understanding".

In order to address this issue, we shall be concerned here with calculations. By a calculation I mean, in effect, the action of a general purpose computer operating according to some computer program. (Strictly speaking, the computer must be idealized in that it has an in principle unlimited storage capacity — which could be in the form of an unlimited supply of floppy discs, say. The appropriate idealized mathematical concept is that of a Turing machine, but it will not be necessary to go into the details of what this term means here.) Another word for a calculation is an algorithm. Operations that can be achieved (in principle) calculationally are referred to as algorithmic. It should be realized that calculations (algorithms) are not merely the performing of ordinary operations of arithmetic, such as just adding or multiplying numbers together, but can involve other things also. Well-defined logical operations can also be part of a calculation. For an example of a calculation, we might consider the following task:

(A) Find a number that is not the sum of three square numbers.

By "number", I mean here a "natural number", i.e. one of

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ....

A square number is the product of a natural number by itself, i.e. one of

0=0², 1=1², 4=2², 9=3², 16=4², 25=5², 36=6², ....

The calculation (A) could then proceed as follows. We try each natural number in turn, starting with 0, to see whether or not it is the sum of three squares. We need only consider squares that are no larger than the number itself. Thus, for each natural number, there are only finitely many square numbers to try. As soon three square numbers are found that do add to it, then our calculation moves on to the next natural number, and we try again to find a triplet of squares (less than the number) that sum to it. Our calculation stops only when a natural number is found for which each such triplet of squares fails to add to it. To see how this works, start with 0. This is 0²+0²+0² so it is indeed the sum of three squares. Next we try 1 and we find that although it is not 0²+0²+0² it is indeed 0²+0²+1². Our calculation tells us now to move on to 2 and we ascertain that although it is not 0²+0²+0² or 0²+0²+1², it is indeed 0²+1²+1²; we then move on to 3 and find 3=1²+1²+1²; then to 4, finding 4=0²+0²+2²; then 5=0²+1²+2²; then after finding 6=1²+1²+2² we move on to 7, but now all triplets of squares (each member no greater than 7)

0²+0²+0², 0²+0²+1², 0²+0²+2², 0²+1²+1², 0²+1²+2²,
0²+2²+2², 1²+1²+1², 1²+1²+2², 1²+2²+2², 2²+2²+2².

fail to sum to 7, so the calculation halts and we reach our conclusion: 7 is a number of the kind we seek, being not the sum of three squares.

However, with the calculation (A) we were fortunate. Suppose we had tried, instead, the calculation:
(B) Find a number that is not the sum of four square numbers.

Now when we reach 7 we find that it is the sum of four squares: $7 = 1^2 + 1^2 + 1^2 + 2^2$, so we must move on to 8, finding $8 = 2^2 + 2^2 + 2^2 + 2^2$, etc. The calculation goes on and on: $23 = 1^2 + 2^2 + 3^2 + 3^2$, $24 = 0^2 + 2^2 + 2^2 + 4^2$, ... and it never seems to stop at all. In fact it never does. It is a famous theorem first proved by the great eighteenth century mathematician Joseph L. Lagrange that every number is, indeed, the sum of four squares. It is not such an easy theorem (and even Lagrange's contemporary the great Swiss mathematician Leonhard Euler, a man of astounding mathematical insight, originality and productiveness, had tried but failed to find a proof). I am certainly not going to go into Lagrange's argument here, so let us instead try something very much simpler:

(C) Find an odd number that is the sum of two even numbers.

I hope that it is obvious that this calculation will never come to an end!

I have given two examples ((B) and (C)) of calculations that never terminate. In the first case this fact, though true, is not at all easy to ascertain, whilst in the second, its non-termination is obvious. Calculations may or may not terminate and, moreover, in the cases when they do not terminate it may be hard to see that they do not, or it may be very easy. We may reasonably ask: by what procedures do mathematicians convince themselves and others that certain calculations do not in fact terminate? Are they themselves following some calculational (or algorithmic) procedure in order to ascertain things of this kind?

The answer appears to be "No". This is essentially one of the implications of Gödel's famous incompleteness theorem, and it is important that we try to understand it. This theorem tells us that no set of rules whatsoever will be sufficient to ascertain correctly, in all cases, that non-terminating calculations do not in fact terminate. Here, by a "set of rules" I mean some system of formalized procedures for which it is possible to check entirely computationally whether or not the rules have been correctly applied. But Gödel's theorem appears to tell us more than this, namely that the insights that are available to human mathematicians — indeed, to anyone who can think logically with understanding and imagination — cannot be completely formalized as such a set of rules. Rules can sometimes be a partial substitute for understanding, but they can never replace it entirely.

In order to see how Gödel's theorem (in the simplified form that I shall give, which depends on some ideas due to Alan Turing) demonstrates this, we shall need a slight generalization of the kind of statement about calculations that I have been considering. Instead of asking whether or not a single calculation, such as (A), (B), or (C) ever terminates, we shall need to consider a calculation that depends on — or acts upon — a natural number $n$. Thus, if we call such a calculation $C(n)$, we can think of this as providing us with a family of calculations, where there is a separate calculation for each natural number $0, 1, 2, 3, 4, ...$, namely the calculation $C(0), C(1), C(2), C(3), C(4), ...$, respectively, and where the way in which the calculation depends upon $n$ is itself entirely calculational. This means that $C(n)$ is the action of some algorithm applied to the number $n$. Think of an ordinary general purpose computer, and regard $n$ as merely providing the "data" for the action of some computer. What we are
interested in is whether this computer action ever stops or not, for each choice of \( n \).

In order to clarify what is meant here, let us consider two examples, slightly generalizing the ones given above:

(D) Find a number that is not the sum of \( n \) square numbers

and

Find an odd number that is the sum of \( n \) even numbers.

It should be clear from what has been said above that the calculation (D) will stop only when \( n = 0, 1, 2, \) and 3 (finding the numbers 1, 2, 3, and 7, respectively, in these cases), and that (E) stops for no value of \( n \) whatever. If we are actually to ascertain that (D) does not stop when \( n \) is 4 or larger we require some formidable mathematics (Lagrange's proof); on the other hand, the fact that (E) does not stop for any \( n \) is obvious. What are the procedures that are available to mathematicians for ascertaining the non-stopping nature of such calculations generally? Are these very procedures things that can be put into a calculational form?

Suppose, then, that we have some calculational procedure \( A \) which, when it successfully terminates, provides us with a demonstration that a calculation such as \( C(n) \) actually does not ever stop. I am certainly not requiring that \( A \) can always decide that \( C(n) \) does not stop when in fact it does not, but I do insist that \( A \) does not ever give us wrong answers, i.e. that if it comes to the conclusion that \( C(n) \) does not stop, then in fact it does not. (If \( A \) did give us wrong answers of this kind, then in principle we could check up on this fact because its error would be recognized as soon as \( C(n) \) comes to a halt.) If \( A \) does not in fact give us wrong answers, we say that \( A \) is sound.

In order to understand how \( A \) could itself act as a calculation, we need to have a way of coding the different calculations \( C(n) \) on which \( A \) acts. All possible different \( C \)'s can in fact be listed, say as

\[
C_0(n), \ C_1(n), \ C_2(n), \ C_3(n), \ C_4(n), \ C_5(n), \ldots
\]

where we can take this ordering as being given, say, as some kind of numerical ordering of computer programs. (To be explicit, we could take this ordering as being provided by the Turing machine numbering given in The Emperor's New Mind, so that then the calculation \( C_q(n) \) is the action of the \( q \)th Turing machine \( T_q \) acting on \( n \).) One technical thing that is important here is that this listing is computable, i.e. there is a single calculation \( C \) that gives us \( C_q \) when it is presented with \( q \), or, more precisely, the calculation \( C \) acts on the pair of numbers \( q, n \) (i.e. \( q \) followed by \( n \)) to give \( C_q(n) \).

The procedure \( A \) can now be thought of as a particular calculation that, when presented with the pair of numbers \( q, n \), tries to ascertain that the calculation \( C_q(n) \) will never ultimately halt. We are going to try to imagine that \( A \) might be a formalization of all the procedures that are available to human mathematicians for validly deciding that calculations never will halt. But it is not necessary for us to think of \( A \) in this way just now. \( A \) is just any sound set of calculational rules for ascertaining that some calculations \( C_q(n) \) do not ever halt.
Being dependent upon the two numbers \( q \) and \( n \), the calculation that \( A \) performs can be written \( A(q, n) \), and we have:

(F) If \( A(q, n) \) stops, then \( C_q(n) \) doesn't stop.

Perhaps you are worrying that \( A \) might be able to decide that \( C_q(n) \) actually does stop and would itself stop as soon as it had made that decision. But that is not what we are asking for \( A \) to do. If, in some way, our procedure comes to the conclusion that \( C_q(n) \) actually does stop, then we must ensure that this conclusion makes \( A \) go into a "loop", rather than output some finite conclusion, so that \( A \) does not in fact stop in the circumstance that \( C_q(n) \) does stop. This is just a technical point, and it does not imply any restriction on what procedures we are considering for ascertaining that calculations do not stop.

Now let us consider the particular statements (F) for which \( q \) is put equal to \( n \). This may seem an odd thing to do, but it is perfectly legitimate. With \( q \) equal to \( n \), we now have:

(G) If \( A(n, n) \) stops, then \( C_n(n) \) doesn't stop.

We now notice that \( A(n, n) \) depends upon just one number \( n \), not two, so it must be one of the calculations \( C_0(n), C_1(n), C_2(n), C_3(n), \ldots \) (as applied to a general value of \( n \)). Let us suppose that it is in fact \( C_k(n) \):

(H) \[ A(n, n) = C_k(n). \]

Now put \( n = k \). We have, from (H),

(I) \[ A(k, k) = C_k(k) \]

and, from (G), with \( n = k \),

(J) If \( A(k, k) \) stops, then \( C_k(k) \) doesn't stop.

Let us try to see whether the calculation \( A(k, k) \) actually stops or not. First, suppose that it does. Then (J) tells us that \( C_k(k) \) does not stop. But, by (I), \( C_k(k) \) is the same as \( A(k, k) \), so it follows that \( A(k, k) \) does not stop after all. This must therefore be the right answer: \( A(k, k) \) does not stop. Moreover \( C_k(k) \), being the same as \( A(k, k) \), does not stop either. We conclude that our procedure \( A \) is incapable of ascertaining that this particular calculation \( C_k(k) \) does not stop even though it does not!

At this point it might be advisable to go over the whole argument again, just to make sure that I have not "put one over" on the reader! Admittedly there is an air of the conjuring trick about the argument, but it is perfectly legitimate, and it only gains in strength the more minutely it is examined. We have found a calculation \( C_k(k) \) that we know does not stop; yet the given calculational procedure \( A \) is not powerful enough to ascertain that fact. This is the Gödel(-Turing) theorem in the form that I require. It applies to any calculational procedure \( A \) whatever, that we know to be a sound, for ascertaining that calculations do not stop. We deduce that no sound set of calculational rules (such as \( A \)) can ever suffice for ascertaining that calculations do not stop. Moreover, since from the knowledge of \( A \) and its soundness, we can actually construct a calculation \( C_k(k) \) that we know to be sound, we deduce that \( A \) cannot be a
formalization of the procedures available to mathematicians for ascertaining that
calculations do not stop, no matter what $A$ is. Hence:

Human mathematicians are not using a knowably sound
algorithm in order to ascertain mathematical truth.

It seems to me that this conclusion is inescapable. However many people
have trouble coming to terms with it, and certainly many have argued against the
stronger deduction that there must be something essentially non-calculational in
our thought processes. Indeed, there are various possible loopholes to consider.
Let us try to go carefully through these in turn.

Some possible viewpoints

Let us accept, on the basis of the preceding argument, that human
mathematicians, when they form their views of the truth or otherwise of
mathematical statements, are not using any knowably sound algorithm. What
procedures might they in fact be using? Let me list a number of possibilities. They might use:

[1] A horrendously complicated unknowable algorithm $X$;
[2] An unsound (but presumably approximately sound) algorithm $Y$;

Let us consider these possibilities in turn.

[1] Do we use a horrendously complicated unknowable algorithm $X$?

An important thing to realize about the insights that allow mathematicians
to come to their conclusions about the truth, or otherwise, of mathematical
statements is that these insights are communicable. Thus, if a certain argument
convinces one mathematician, then the same argument can also be used to
convince any other mathematician — at least in principle, but with (admittedly)
considerably varying amounts of difficulty. It should always be possible to break
down a mathematical argument into steps that themselves represent things that are "obvious" to all, rather than being things that are personal to particular
individual mathematicians. The same comment would apply to the "Gödelian
insight" that allows one to deduce from a belief in the soundness of a procedure
for ascertaining that certain calculations do not stop (such as $A$ above) that a
particular calculation (such as $C_k(k)$ above) does not in fact stop. It is not that we
are talking about different possible horrendous algorithms that might happen to
be running around in different mathematicians heads, depending upon how
each one's brain might happen to be "wired up". We are talking about one
horrendously complicated unknowable algorithm $X$ that applies universally to
the mathematical community as a whole.
It is true that a good deal of mathematics is often carried out within what are called "formal systems", and these are essentially algorithmic structures, but these cannot encompass the entirety of the insights that are available to mathematicians, as Gödel's procedure demonstrates. Such formal systems are never so complicated that they are actually "unknowable", but some people might consider that, in certain cases, their meanings are so esoteric that their soundness is unknowable. However, any system whose soundness cannot be trusted cannot itself be used without qualification in a mathematical argument. An argument based on a questionable formal system \( F \) would have to run something like "Assuming \( F \), then it follows that ..." if it is to be acceptable to the mathematical community as a whole. I do not see how any such \( F \) could possibly encompass precisely the insights that are available to the mathematical community. If \( F \) is accepted as sound, then the Gödel procedure yields a mathematical proposition \( G \) that must also be accepted as sound, yet which is inaccessible by the methods of \( F \). (Here, \( F \) plays the role of \( A \), and \( G \) is essentially the assertion that \( C_k(k) \) does not terminate.) If \( F \) is not accepted as sound, then we cannot use it (without qualification) for ascertaining the truth of mathematical propositions. Either way, \( F \) cannot precisely encompass all the insights available to human mathematicians.

It seems to me that a similar kind of argument would apply to our putative unknowable algorithm \( X \). Can it be that, unknown to all of us, when mathematicians come to their conclusions about the truth of mathematical statements they are just acting according to the horrendous \( X \)? This seems to be totally at variance with what mathematicians seem actually to be doing when they express their arguments in terms that can (at least in principle) be broken down into assertions that are "obvious", and agreed by all. I would regard it as far-fetched in the extreme to believe that it is really the horrendous unknowable \( X \), rather than these simple and obvious ingredients, that lies lurking behind all of our mathematical understanding.

There is another argument that, in my view, adds convincing support to the belief that mathematicians do not actually use \( X \). For how is it that \( X \) could have arisen by natural selection? One need but glance at the contents of any respectable mathematical research journal to see how far removed from everyday experience are the subject-matter and lines of reasoning that are used by mathematicians. If \( X \) is needed for doing all this, and if \( X \) is shared among all mathematicians — and, in principle, among all thinking people — then somehow \( X \) would have to have arisen by natural selection, the process that, over thousands of centuries, has made Man what he is. Yet the selective value of a potential ability to do such sophisticated mathematics must have accounted for nil throughout Man's history! Even today, there is very little direct survival value related to an ability to reason accurately with obscurely defined infinite sets. If such an ability depended upon the possession, within the heads of at least a significant proportion of us, of the horrendously complicated unknowable \( X \), then one is faced with the apparently insuperable problem of comprehending how anything like \( X \) could have arisen by natural selection.

In my own view, what underlies an ability to do mathematics, whether sophisticated or not, is something very different from the putative algorithm \( X \); it is our general (non-algorithmic) ability to understand. What natural selection has provided is an ability to understand one's surroundings and to reason about
the implications of one's actions. Accordingly, Man's superb ability to survive must, in large measure, have arisen from his ability to understand. This ability would have enabled him to achieve an effective superiority over his less comprehending adversaries. It would have had a profound value to him for his survival in many different ways, such as in the construction of mammoth traps and the like. The quality of understanding is, perhaps fortuitously, also the crucial quality that is needed for doing mathematics. It was not an algorithm X that was favoured, in Man (at least) by natural selection, but this wonderful ability to understand! The Gödel argument shows that mathematical understanding, at least, is not an algorithmic activity, and it seems to me that we must accept that a non-algorithmic quality, namely an ability to understand, can indeed arise by natural selection.

[2] Do we use an unsound (approximately sound) algorithm Y?

What about the possibility that mathematicians might use an algorithm Y that is only approximately sound? Mathematicians, after all, do make mistakes — not infrequently, in fact (and I write from personal experience here). It would appear that Alan Turing himself may have believed that this was the loophole in the Gödelian argument, when it is applied to the algorithmic action that he himself apparently believed applied to the actual processes taking place in the human brain. For myself, I find this possibility to be unlikely as the real explanation. Many of the above objections to X, it seems to me, would apply also to Y. Moreover, although mathematicians do indeed not uncommonly make mistakes in their personal workings and in their guesses as to the truth, it is rare for these errors to survive significantly in their published writings. The essential point is that the errors are recognizable as errors. When a mistake has been pointed out, either by someone else or else by the same mathematician at a later time, it is seen to be an error — or at least if, for some reason a mental blockage persists, such a blockage could in principle be removed by further explanation and insight. The way that mathematics progresses is not at all like the blind following of an unsound algorithm Y.

Moreover, unsound algorithms are liable to land one in contradictions. In particular, in the arguments given above, if the algorithm A were unsound, then it would incorrectly assert that some calculation \( C_q(n) \) does not stop when indeed it does. If the calculation \( C_q(n) \) is actually stumbled upon and performed, and is seen to come to an end while \( A(q,n) \) successfully terminates, then we would know that something is amiss. Of course things like this do occur with working mathematicians: errors in an argument may be spotted when a counter-example to the conclusion is found. But that would seem to be something very different from what is being envisaged here. Actual errors in a mathematical argument could, in principle, have been spotted in any case. The counter-example merely provides a route to finding something that could also be found directly, namely by checking through all the details of the original argument. In the case of Y, this putative algorithm is supposed to represent the totality of all the means that are available to human mathematicians. Thus Y is in principle not correctable by human reasoning, since whatever means are available for achieving this should already have been included in the procedures of Y. (I shall return to the issue of possible self-improving algorithms under the next heading.)
There is a type of procedure that is sometimes referred to, in the context of AI, as "heuristics", which should be included under the present heading. Heuristics refer to "judgements" or "rules of thumb" that may be included in a computer program, but that do not provide definitive mathematical solutions to the problem at hand. For example, in a chess-playing computer program, the computer may be instructed to analyse positions by calculating the results of all possible alternative successions of moves by each player, a good many moves deep. It would not be possible to take this analysis as far as checkmate (or as far as some accepted winning position) in most cases, there being far too many alternatives to work through before an adequate depth of move can be achieved for this. Instead, the various strands of computation would be terminated as soon as some appropriate "heuristic" criterion is satisfied. The resulting program would not play perfect chess, but nevertheless it could be made to play chess extremely well, and some such programs — most particularly "Deep Thought" constructed by a group at Carnegie Mellon headed by Hsiung Hsu — can defeat all but a very small minority of human players.

Computers that act according to such procedures are still acting algorithmically, of course, even though their action is not given by some algorithm that is guaranteed to provide the desired result (e.g. to play perfect chess). If such procedures were applied to the solving of mathematical problems as a whole, then they could be thought of as giving an algorithm such as Y, and it might be imagined that this is indeed how mathematicians actually operate. However, a known procedure of this type would certainly not be accepted by the mathematical community as providing an adequate criterion of mathematical proof. Mathematicians require a degree of rigour that makes such heuristic arguments unacceptable — so no such known procedure of this kind can be the way that mathematicians actually operate. We are left with the possibility that Y might be a horrendously complicated unknowable algorithm, and we are really back with the situation that we had before, with the unknowable algorithm X (but with Y as actually an inferior inaccurate version of X), and the same objections will apply as before. Most particularly, the arguments from natural selection are at least as damning as they were before.

[3] Do we use an ever-changing algorithm?

There is, however, the possibility that we might be using an algorithm, perhaps like X or Y, but that is continually changing with time by being improved with experience. But now we must ask, as we did briefly before: by what procedures is our putative algorithm supposed to improve itself. If these very procedures are things that could be programmed on a computer, then they are themselves algorithmic. All the arguments that we have been considering before should really be applied to the entire algorithm controlling the computer's action, including that part which governs its mode of change. This algorithm includes all the "learning procedures" that have been laid down at the beginning. If the only ways in which the algorithm can change is by such pre-assigned algorithmic means, then we do not, strictly speaking, have a changing algorithm at all but just a single algorithm, say X or Y, as before and all the objections raised above will still apply.
There is a particular type of learning system that has gained considerable popularity in recent years, referred to as a neural network. This type of system is based initially on certain ideas about how brains actually work — most particularly, the idea of brain plasticity, according to which the strengths of the various connections between neurons in the brain are continually being modified in order to improve performance — but in a "neural network" the system is implemented entirely electronically. At the present time, such systems are normally run on ordinary general-purpose electronic computers, but the intention is that eventually, special-purpose electronic hardware would be constructed, so that a much greater speed and efficiency can be achieved. (It should be mentioned that the biological mechanisms whereby the strengths of connections between neurons in the brain are actually modified are largely unknown. Neural networks use their own procedures, and these probably differ in important ways from what happens in the brain plasticity of actual brains.) Neural networks have been applied to many varieties of different task, such as the recognizing of faces, or distinguishing the quality of different sounds, or prospecting for mineral deposits, or controlling a robot arm. They can be effective for tasks where understanding seems not to be playing a crucial role — so it is hard to see that they could be of much use in forming mathematical judgements.

A basic feature of a neural network, distinguishing it from the normal type of algorithmic programming, is that rather than having a previously given algorithm that is specifically provided in order to solve some particular problem or class of problems (which would sometimes be referred to as the normal "top-down" organization) we are, instead, initially provided with merely a loosely connected family of units (the electronic "neurons") where the strengths of the various connections are continually being modified in order to maximize the quality of the output. In this way the system continually "learns", so that the output improves all the time. The action of such a system is still algorithmic, since it can be implemented on an ordinary general-purpose electronic computer, but there is a crucial difference in the intentionality underlying the algorithm. Now, the algorithm according to which the system acts at any one time is not one that has been fed in initially in order to provide a specific solution to a pre-assigned problem, but it is one that has gradually evolved on the basis of improving the quality of its output ("bottom-up" organization).

Nevertheless, the procedure taken as a whole still constitutes an algorithm, provided that the judgement as to the quality of the output at any given stage is made according to some (possibly "heuristic") algorithmic criterion. This would indeed normally be the case if one is thinking of using the neural network to solve some clearly specified problem. Certainly, if one attempted to use a neural network to solve problems in pure mathematics, then its action would be entirely algorithmic. Thus, we would again have a situation of the type already considered under the previous heading: an approximate algorithm $Y$; and the various objections to this being actually the procedure that mathematicians use would apply as before.

[4] Is the role of the environment essential?

Of course, it might be the case that it is the continual input that we get from the environment that gives our mathematical understanding its (apparently?)
non-algorithmic character. It is undoubtedly the case that mathematicians are indeed being continuously influenced by their environment. Even the very idea of what we mean by a natural number could hardly have formed in our minds were it not for our experiences with various different manifestations of such numbers — at least of the smaller ones: "three oranges", "two socks", "ten pennies", etc. Moreover, although advanced mathematics need not have any very precise correspondence with our immediate experiences, and mathematical reasoning is wholly abstract, mathematicians frequently do gain their inspirations from everyday things in the world around them.

But is this input from the environment something that has an essentially non-algorithmic character? To put the question another way, might it be that it is in principle impossible to simulate the environment computationally? The question is not whether or not the actual environment of a particular individual could be precisely simulated, but whether or not everything that might be relevant to the building up of an individual's mathematical understanding could be computationally simulated. I find it hard to see where there can be any essentially non-algorithmic input coming in here. In any case, if one takes refuge in the possibility that it is the environment that supplies the needed non-algorithmic ingredient in our mathematical thinking, then one is left with the problem of explaining how the physical environment can behave non-algorithmically. If one is prepared to believe that the physically governed external world could behave in a way that cannot be simulated computationally, then one must surely also accept that a (physically controlled) human brain might behave non-computationally. This, after all, is what the thrust of my argument is intended to show.

[5] Do random ingredients change the discussion?

In the discussions so far I have been considering "algorithmic action" to be essentially that which could be carried out in principle by a by a modern general-purpose electronic computer (with an unlimited store) — which, technically, means by a Turing machine. The action of a Turing machine is always completely determined in advance, even though the ultimate behaviour of the machine might be hard to predict. One might expect that some essential differences could arise when we relax this condition, and allow the presence of random ingredients to be introduced into the calculations. In modelling the environment, in particular, one would normally expect that there would be many uncontrollable parameters, and these would have to be modelled in some way in terms of a random input. Moreover, in the action of the brain itself, it is to be expected that there would be a great deal of random activity, and one could imagine that this could have important implications with regard to its overall performance.

However, in computational simulations of physical systems, the presence of random ingredients poses no problem, provided that one is not asking for a prediction of the actual detailed way in which a particular physical system is going to behave, but is merely asking for a typical example of the way in which such a physical system might behave. The latter, after all, is all that we would be asking of an artificially intelligent system. There are many algorithmic means of producing an effectively random input. In place of a "genuinely" random
sequence of digits, the computer produces a sequence of what are called "pseudo-random" digits which, for all intents and purposes, behaves as a random sequence would, despite the fact that the sequence is generated entirely algorithmically. Thus, in practice, it does not seem to be any significant restriction to limit one's attention to entirely algorithmic computation.

In present-day physics, there are two distinct ways in which (apparently) random behaviour can result in a seemingly well-specified physical system. In the first place, there is the phenomenon known as chaos, whereby a completely deterministic classical system can, in practice, behave as though it were not deterministic at all. This is because the accuracy according to which the initial state needs to be known, for a deterministic prediction of its future behaviour, can be totally beyond anything that is conceivably measurable. An example that is often quoted in this connection is the detailed long-range prediction of the weather. The laws governing the motion of air molecules, and also the other physical quantities that might be relevant, are all perfectly well known; however the weather patterns that may actually emerge, after only a few days, depend so subtly on the precise initial conditions that there is no possibility of knowing these conditions accurately enough for reliable prediction. Of course the number of parameters that would have to enter into such a calculation would be enormous, so it is perhaps not surprising that prediction, in this case, might prove to be virtually impossible in practice. However, such so-called chaotic behaviour can occur also with very simple systems consisting of only a small number of particles. Imagine, for example, that one is to pocket the fifth snooker ball E in a chain A, B, C, D, E by hitting A with the cue so that A hits B, causing B to hit C, then C to hit D, and finally D to hit E into the pocket. So long as no two successive balls are initially close up against one another, the accuracy needed for this is far in excess of the abilities of any expert snooker player. If there were ten in the chain, then even if the balls were perfectly elastic perfect spheres, the task of potting the final ball would be far beyond the most accurate machinery of modern technology. In effect, the behaviour of the later balls in the chain would be random, despite the fact that the Newtonian laws governing the behaviour of the balls is mathematically completely deterministic.

From time to time, it has been suggested that this phenomenon of chaos, if it occurs in the internal action of a physical brain, might be what enables our brains to behave in ways that appear to differ from the computably deterministic activity of a Turing machine. However, if all that we can get from chaos is randomness, then this will be no use to us. For randomness itself does not give us anything usefully beyond ordinary algorithmic computation. Does chaos give us anything that is usefully non-computable, rather than just randomness? In fact the behaviour of chaotic systems can sometimes be highly structured and much more interesting-looking than purely random behaviour. It remains to be seen, however, whether this kind of structure contains anything approximating genuinely non-algorithmic behaviour. It would be very interesting to see if this is possible, but so far no-one appears to have argued any kind of case that it is.

The other place where modern physics asserts that randomness arises in the precise behaviour of a physical system is in quantum mechanics. Many people are familiar with the fact that quantum theory does not give us a deterministic picture of the time-evolution of the world and that it merely provides probability values for the various alternative outcomes of experiments that may be
performed on a system. What is not so commonly appreciated is that this probabilistic — i.e. random — behaviour does not occur at the submicroscopic quantum level of molecules, atoms, and subatomic particles. For at that level, things are described as evolving according to the precise deterministic Schrödinger equation, and the behaviour is just as deterministic as it is at the classical level of everyday experiences, where the laws of Newton and Maxwell (and also Einstein) hold sway. Instead, the quantum randomness is effectively taken to occur at some mysterious middle-ground level, intermediate between the quantum and classical levels. A feature of the quantum level is that all the various alternative things that a system might do have also to be allowed to happen together in some mysterious kind of quantum superposition. But when the effects of some of these superposed alternatives become magnified to the classical level, in accordance with a measurement being performed on the system, one alternative or the other becomes realized as actuality. At this stage, Nature's choice of alternative is governed merely by probability. According to theory, what one must do — and this is all that quantum mechanics tells us about what happens during measurements — is to adopt the strange procedure referred to as state-vector reduction (or wavefunction collapse). According to this, the quantum description (the state-vector, or wavefunction) undergoes a discontinuous random change, where what happens is restricted only by the probability laws that the theory tells us how to calculate. It is here, and only here, that the the deterministic character of the underlying theory is changed to a probabilistic one. At both the quantum level (Schrödinger's equation) and the classical level (Newton-Maxwell laws) the behaviour is basically deterministic (considerations of chaos notwithstanding), and the randomness comes in only when one tries to bridge the gap between the two. In this connection it should be pointed out that the difference between the quantum and classical levels is not really one of size. (Quantum effects have been observed to take place over a separation of many metres.) It would be somewhat more accurate to say that the quantum level refers to phenomena for which the differences in energy between the states under consideration are small. In fact the whole issue of what actually happens when the alternative effects of differing quantum possibilities are magnified to the classical level whenever such a "measurement" takes place is fraught with controversy. But whatever one believes quantum theory is telling us about the actuality of the world, some procedure akin to the one just described must be adopted in order that the necessary probability calculations can be made.

What might be the relevance of this to the activity of a human brain? As was the case with chaos, all that we appear to obtain, when the procedures of quantum theory are adjoined to the mechanistic determinism of classical theory, is a random element that would seem to be of little use to us if we are expecting to find something usefully non-computable coming out of it all. (In fact, there are certain profoundly intriguing features of the behaviour of quantum systems that differ significantly from what can be achieved by classical systems — such as non-locality, and the possibility that certain calculations can, in principle, be performed more rapidly — but these do not change the class of problems that can or cannot be solved from what they were for a Turing machine.) As our understanding rests at the moment, the incorporation of conventional quantum mechanics into our discussion does not significantly help us in our search for a non-algorithmic ingredient that could be relevant to our thought processes.
Thus, as things stand, chaos and quantum mechanics both leave us with only randomness and not with something usefully non-algorithmic. There remains a hope, in each case, that future developments in theory might improve the situation. With chaos, this hope rests on the possibility that something more useful than mere randomness might arise in suitable circumstances; with quantum mechanics, it rests on the finding of a new physical theory with a more satisfactory way of handling the transition between the quantum and the classical level. Both hopes represent highly unconventional viewpoints at the present time. I shall return briefly to these possibilities under the final heading.

[6] Must we resort to mysticism?

Since we seem, so far, to have been led into an impasse by following the methods of conventional scientific theory, is it not perhaps reasonable that we should abandon the physicalist view of the mind altogether and turn, instead to a viewpoint more in line with some form of mysticism? Even Kurt Gödel appears to have found himself driven to a rather mystical standpoint. In commenting on Alan Turing’s computational view of the mind, Gödel suggested that the mind need not be limited by the brain’s finiteness. He seemed to have believed, along with Turing, that the objects of the physical world should behave "mechanically" — by which he meant, in effect, "algorithmically" — but unlike Turing, he appeared to believe that the human mind was something beyond the physical brain. In this way, the mind would be able to perceive the Platonic truth of mathematical statements in some way not totally dependent on the physical action of the brain.

As I have stated in my introductory remarks, I intend to stay within the framework of scientific possibilities in addressing the issues of mind. The evidence that mental attributes result from activity in the physical brain seems to me to be overwhelming. While it is certainly true that we do not understand how mental phenomena can arise from the activity of a physical object, the case is surely powerful that somehow they can, at least when the "physical object" is a living human brain. To my own way of thinking, the resolution of the profoundly puzzling issues raised by the evident fact that certain types of physical object can indeed give rise to conscious minds must ultimately lie in our finding a deeper understanding of what physical objects actually are. As science has progressed, our descriptions have become more and more accurate and complete; but at the same time, they have become more and more abstract and mathematical. Our very picture of concrete reality has changed to something almost totally abstract and, in many ways, mysterious. It seems that we need to turn to the abstract Platonic world of mathematical forms if we are to find where ultimately concrete reality is to find its home. The mystery of mind can ultimately be understood only in terms of the mystery of matter itself, and in the undoubted mystery of the mathematics that so precisely seems to control its every action. This is not mysticism; it is the ultimate purpose of science itself.

[7] Are there non-algorithmic physical laws?

If we are to stay within the laws of physics in our search for a non-algorithmic action that might underlie our conscious thought processes (such as those that give rise the mental quality of mathematical understanding), then we
must find something usefully non-algorithmic in those laws. As I asserted earlier, mere randomness will not do. We need something a good deal more sophisticated than randomness if we are ever to find a possible physical basis for our conscious understanding.

What kind of physical laws could exhibit non-algorithmic behaviour? It is certainly possible to provide various different kinds of "toy model universe" that are non-algorithmic. Such a model can even be completely deterministic, as the following (admittedly totally unrealistic) example illustrates: Let the state of our "universe" be given by a pair of natural numbers \( n, q \), where the (discrete) time is also given by a natural number \( t \). The rule for the evolution of this universe is that, if the state at time \( t \) is \((n,q)\), the state at time \( t+1 \) is to be \((n+1,q)\) if the calculation \( C_n(n) \) stops and it is \((q+1,n)\) if \( C_n(n) \) doesn't stop. Since there is no general algorithm for deciding whether a calculation \( C_n(n) \) stops or not, the future behaviour of our model universe is not computable, even though it is completely deterministic. Although this particular model is not to be taken seriously as a picture of reality, of course, it is not at all impossible that the actual physical laws could exhibit a non-algorithmic behaviour of this general kind.

There are many classes of mathematical problem that are non-computable and that could take the place of the non-stopping of \( C_n(n) \) in the example just cited (the solubility of families of polynomial equations in terms of natural numbers, the topological equivalence problem for four-dimensional manifolds, the problem of deciding which finite sets of polygonal tiles will cover the Euclidean plane without overlaps or gaps, etc.). There could also be other kinds of example of a much more subtle, and possibly more realistic, nature.

There is, as yet, no indication that the phenomenon of chaos can exhibit anything usefully non-random that approximates a non-algorithmic action of this general kind. In this connection it should be pointed out that chaotic behaviour can certainly be simulated computationally, and I think that the arguments that I have given earlier argue strongly against such a computational model of conscious understanding. For reasons such as these, I think that it is unlikely that chaos can supply our needed answer.

On the other hand, there are numerous reasons, coming from physical theory itself, for believing that present-day quantum theory must someday be profoundly modified, the procedure of state-vector reduction being distinctly unsatisfactory physically. It is my own opinion that such a necessary modification will someday be discovered, and that it is here that we shall find our needed non-algorithmic behaviour - to replace or amend the probabilistic procedures that present-day quantum theory seems to require. Such a theory would need to have an even deeper, more profoundly subtle, and universal nature than the theories that science has provided to date, and its unity with the Platonic world of mathematics should also be more secure. Perhaps, then, we shall be better able to find a basis for mental phenomena within scientific explanation than has been remotely possible hitherto.
Notes

1. These four alternatives are explicitly described in, for example, Johnson-Laird (1987), p.252. (It should be noted, however, that what he refers to as the "Church-Turing thesis" is not given altogether appropriately, and its relevance to "viewpoint C" requires further discussion. For a spectrum of opinions on the mind-brain-computer issue, see also various other articles in Blakemore and Greenfield (1987).


3. See the discussion in Moravec (1988), Chapter 6.

4. From his explicit writings, I take this to be the originally stated viewpoint of Searle (1980).

5. See, for example, Searle (1980).

6. For earlier arguments in support of the thesis that Gödel's theorem shows that (mathematical) thinking cannot be computational, see Nagel and Newman (1958), Lucas (1961). There have been numerous counter-arguments to these; see, for example, Benacerraf (1967), Lewis (1969, 1989), Hofstadter (1979).

7. See Bell (1987).

References


Physics at the Large Hadron Collider LHC

G. Flügge

III. Physikalisches Institut, RWTH Aachen, Germany

Abstract

During this decade high energy physics will seek to penetrate into the multi TeV energy regime. This article gives a brief summary of the physics motivation for this step and a short description of the 16 TeV collider LHC, which is presently being planned at CERN. The main section of the article highlights some of the physics phenomena which were studied in great detail at the LHC workshop in Aachen.

1 Introduction

To build the next big machine in high energy physics in Europe requires an investment of a major fraction of capital resources and manpower available in Europe over a period of about one decade. For many of us, at least for the older ones like me, it may be the last machine they will work on.

Consequently, the physics motivation for such a machine has to be convincing. First of all it has to be able to answer those questions which we believe can be clearly formulated in the framework of our standard model, i.e. questions related to the top quark, the τ neutrino, the origin of CP violation and of course last not least, the origin of symmetry breaking and of masses. However, in addition to this rather clearly formulated programme the machine has to have a large discovery potential towards new phenomena. Here, questions can only vaguely be formulated in the context of speculative ideas beyond the standard model.

The outline of my talk will thus be as follows. After a short physics motivation and a few words about the LHC machine I will cover the physics subjects wich in my opinion are the most important ones and provide the best arguments for the LHC, search for the standard Higgs boson, heavy quark physics including CP violation and neutrino physics. I will close with a few remarks on physics beyond the standard model and on logistics of the LHC machine and detectors.

A word of apology is in order here. Instead of refering in detail to all contributors of the LHC Workshop in Aachen [1] last year - and the references of the contributors - on which this talk is based, I just refer to the convenors of the pp physics session, on whose talks I heavily lend, G. Altarelli [2], D. Denegri [3] and F. Pauss [4].

2 Physics Motivation

The basic argument why the next step in high energy physics will come from a machine in the energy range of a few TeV is rather simple and reminds of similar arguments that predicted massive bosons in the framework of the Fermi theory of weak interactions. In the standard model let us consider the cross section of elastic scattering of longitudinal gauge bosons, e.g.

\[ \sigma(W_L W_L \rightarrow W_L W_L) = \frac{G_{F}^{2}}{16\pi\sqrt{2}} \sim E_{CM}^{2} \]
This cross section diverges quadratically in energy. If one inserts numbers one finds that for this simple process the standard model prediction will get in conflict with the unitarity limit at \( \sqrt{s} \approx 1 \div 2 \) TeV. Thus, something has to happen in this energy range. There could either be the Higgs boson

\[
\sigma(W_LW_L) = H^0
\]

which would damp the cross section or something else we don’t really have an idea of.

The argument has recently been worked out from a more general point of view. The result of this study is summarized in Fig. 1. The argument is roughly speaking as follows: if one studies the renormalization equation related to the couplings of the Higgs potential

\[
V = \mu^2 |\phi|^2 + \frac{\lambda}{2} |\phi|^4
\]

one realizes, that two kinds of limits occur. For our argument, the most important limit is the singularity at which \( \lambda(\mu) \) grows beyond all borders if \( \mu \) is not bound to be smaller than a cutoff energy \( \Lambda : \mu \leq \Lambda \). This is represented by the horizontal lines in Fig. 1. Consistency of the standard model up to a given cutoff \( \Lambda \) thus implies upper bounds on the Higgs mass. The important message from Fig. 1 is that if consistency within the standard model is required either \( m_H \) has to be rather small (\( m_H \approx 200 GeV \)) in which case the standard model could be extended up to the Planck mass, or, if \( m_H \approx 500 GeV \), new physics will come in at the level of the cutoff energy of the order of 1 TeV. For completeness I note in passing, that the vertical limits (vacuum stability) are derived from the requirement \( \lambda(\mu) \geq 0 \) for \( \mu \leq \Lambda \).

3 The LHC Machine

If we convince ourselves that something new will occur in the TeV range, we have to make up our mind how to reach it. The choice is either an \( e^+e^- \) collider or - for the same constituent energy - a 10 times more energetic pp collider. Although many physics arguments are in favour of an \( e^+e^- \) machine the technique to build a 2 TeV \( e^+e^- \) collider (CLIC) considered in La Thuile [5] is still out of reach in the near future and the CERN management decided to head for a 16 TeV pp collider in the LEP tunnel, the Large Hadron Collider LHC. Thus the CERN management came forward with a definite proposal for a Large Hadron Collider which will have three options given in Table 1.

<table>
<thead>
<tr>
<th>CM energy</th>
<th>max. luminosity/cm(^{-2})s(^{-1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>proton-proton collisions</td>
<td>( \leq 16) TeV</td>
</tr>
<tr>
<td>lead-lead collisions</td>
<td>1300 TeV</td>
</tr>
<tr>
<td>electron-proton collisions</td>
<td>1.7 TeV</td>
</tr>
<tr>
<td></td>
<td>1.3 TeV</td>
</tr>
</tbody>
</table>

The principle layout of the machine which makes use of most of the existing machine infrastructure at CERN is shown in Fig. 2.

Fig. 3 shows the position of the superconducting magnets of the machine in the existing LEP tunnel on top of the LEP machine. Fig. 4 shows the mockup of a tunnel section.
The main component of the machine will be superconducting dipoles. Fig. 5 shows a cross section of the 'two-in-one' magnet in which both coils and vacuum tubes of the two beams are contained in one iron structure. The superconducting coils of NbTi will be operated at 1.8 K.

A full size prototype (which is however built using the HERA superconducting cable) is presently under test in Saclay (Fig. 6).

4 Physics at the LHC

In this talk, I will concentrate on the pp option of the LHC. Fig. 7 shows a compilation of the most important cross sections in pp collisions. Without going into too much detail, the Figure is shown to illustrate the difficult experimental environment at LHC and to structure this section of the talk. The main problem of pp physics in the TeV range becomes obvious if one compares the total cross section of 100 mb with the cross section for Higgs production of \( \sim 1 \) pb. Interesting events have to be found in a 'background' which is 11 orders of magnitude higher than the process under consideration. The problem becomes even clearer if one looks at the right hand scale of Fig. 7, where the event rates at an expected luminosity of \( 10^{34} \text{ cm}^{-2} \text{ s}^{-1} \) are given. Whereas the total rate is a frightening \( 10^9 \text{ s}^{-1} \) (with even \( \approx 20 \) events overlapping in one burst every 15 ns), the rate of events we are finally looking for is only \( 10^{-2} \text{ s}^{-1} \). Out of these, as we will see below, only a fraction of about \( 10^{-2} \) can be used, so that we are left with some 100 events per year to be filtered out of \( 10^9 \) events/s.

In the following I will pick up from the vast variety of physics processes which could be studied at the LHC a few topics which I consider to be important, namely, in inverse order of cross section, the search for the standard Higgs particle, top quark physics and beauty physics with emphasis on CP violation. I will end with some remarks on \( \tau \) neutrinos.

4.1 Search for the standard model Higgs

In pp collisions the standard model Higgs particle \( H^0 \) is mainly produced through gg fusion. The diagrams for this and other production processes together with the cross sections are shown in Fig. 8. Cross sections are fairly well known within theoretical uncertainties of about a factor 2 which are mainly due to uncertainties in the top mass, parton distributions and QCD corrections.

Besides the dominating cross section due to gluon-gluon fusion also the W, Z bremsstrahlung process turns out to be important for the Higgs search at low Higgs masses.

Fig. 9 shows the decay width of the Higgs particle for different decay modes as a function of the Higgs mass. For masses \( m_H \geq 2m_Z \) the width is completely dominated by Z, W decay. Below \( 2m_W \) the decay \( H^0 \rightarrow \gamma \gamma \) may provide a useful signature for Higgs search. All other decay modes are very difficult to disentangle from multijet backgrounds and will not be considered.

Table 2 summarizes the signatures that have been studied and turned out to be promising for Higgs searches in the various mass ranges.

The decay channel of the Higgs which is best studied is its decay \( H^0 \rightarrow ZZ \) or \( ZZ^*, Z^+Z^- \) with subsequent decay \( Z^0 \rightarrow \gamma \gamma \rightarrow 2\mu \). The total cross section for this channel and the rates expected at a luminosity of \( 5 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1} \) are shown in Fig. 10.

As can be seen in Fig. 11 this channel will produce very clean signals in the mass range of several 100 GeV. The situation becomes more complicated at energies above 700 to 800 GeV, because the signals get very broad (see Fig. 9) and below 130 GeV, where the process runs out of rate.

This situation is studied in more detail in the following Figures. Fig. 12a shows the decay of Higgs particles with different masses compared with various background sources which have been studied. In this Figure perfect resolution of the detector is assumed. The background is dominated by \( tt \) production which nearly everywhere overwhelms the signal. Fig. 12b shows the effect of applying the contraint \( Z^0 \rightarrow \mu \mu \) in a typical iron
Table 2: $H^0$ signatures

<table>
<thead>
<tr>
<th>$H^0$ signature</th>
<th>BR</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_{H^0} \geq 2m_z$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H^0 \to WW \to l\nu jj$</td>
<td>$\sim 20%$</td>
<td>t\bar{t} background</td>
</tr>
<tr>
<td>$H^0 \to ZZ \to l\nu l\nu$</td>
<td>$\sim 0.8%$</td>
<td>$l = e, \mu$</td>
</tr>
<tr>
<td>$H^0 \to ZZ \to llll$</td>
<td>$\sim 0.14%$</td>
<td>e difficult</td>
</tr>
<tr>
<td>$H^0 \to ZZ \to \mu\mu\mu\mu$</td>
<td>$\sim 0.035%$</td>
<td>best signature low rate</td>
</tr>
<tr>
<td>$m_{H^0} &lt; 2m_z$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H^0 \to ZZ^*, Z^<em>Z^</em> \to \mu\mu\mu\mu$</td>
<td></td>
<td>no Z constraint</td>
</tr>
<tr>
<td>$H^0 \to \gamma\gamma$</td>
<td></td>
<td>need very good $\gamma$-detector</td>
</tr>
</tbody>
</table>

toroid detector ($\delta p/p = 12\%$) The $t\bar{t}$ background is considerably reduced and the signal sticks out clearly from the remaining background, which is now dominated by continuum production of $ZZ$ pairs. The width of the signal is affected only at low masses.

For Higgs masses below $\sim 200$ GeV the situation becomes less favourable for the $4\mu$ channel, as shown in Fig. 13a. Since the $Z \to \mu\mu$ constraint can now be applied only for one of the $Z$'s, the background remains sizeable. Also, the Higgs rate gets very low in the mass range of the dip in $\sigma \cdot$ BR of Fig. 10 and below $\sim 130$ GeV where the process finally runs out of rate due to the threshold in BR ($H^0 \to ZZ$). The signal to background ratio could be largely improved if isolation of the $4\mu$ from jets could be required. The effect of this estimated background reduction by a factor of about 50 is shown in Fig. 13b.

Several reactions and Higgs decay modes have been studied to extend the mass range in which the Higgs could be seen. The mass ranges which could be covered by various processes are shown in Fig. 16. Particular attention has been paid to the 'low mass gap' between 130 GeV and the Higgs mass of about 80 GeV up to which LEP 200 experiments would still be sensitive. The most promising reaction which could do the job would be $H^0$ production or $H^0Z$ or $H^0W$ production which has still a large cross section at these $H^0$ masses (see Fig. 8) with the decay $H^0 \to \gamma\gamma$. The Higgs search in this channel would however require $\gamma$ detection with high resolution ($\delta E_\gamma / E_\gamma \leq 1\%$, $\delta \theta \leq 5$ mrad) at highest luminosity which will be a formidable task for an LHC detector.

Fig. 14a shows $\sigma \cdot$ BR and rates expected at a luminosity of $10^{34}$ cm$^{-2}$ s$^{-1}$ for $H^0$ production with subsequent decay $H^0 \to \gamma\gamma$. About 4000 events/year could be expected. Fig. 14b shows the signal over background for a Higgs mass of 120 GeV assuming an e/m calorimeter with an energy resolution of $\delta E_\gamma / E_\gamma = 1\%$ and an angular resolution of 5 mrad. The background contains only the irreducible gg and $q\bar{q}$ fusion processes. The ratio signal/√background would rise from about 5 at $m_H = 80$ GeV to about 11 at $m_H = 150$ GeV.

The background can be strongly reduced if Higgs production from the $W, Z$ bremsstrahlungsprocess

$$pp \to W(Z)H^0 \to l\bar{l}\gamma\gamma$$

is studied. Since however also the signal is reduced to $\sigma \cdot$ BR $\approx 0.0004$ pb the significance stays about the same as in the inclusive $H^0 \to \gamma\gamma$. Fig. 15 shows a simulation of Higgs production from bremsstrahlung together with the irreducible background of the $W\gamma\gamma$ continuum.

4.2 Top Physics

The issue here is of course again the discovery although there are good chances that the top quark will be discovered earlier at Fermilab. Even then strong interest will remain in a good mass determination and the study of decay modes, e.g. into hypothetical charged Higgs particles.
The production in the mass range of $m_t \leq 200$ GeV (upper limit suggested by neutrino scattering and recent LEP data) is dominated by gluon fusion.

As shown in Fig.17 production rates will be very large, of the order of $10^6$/year. Given the latest lower mass limit of $m_t \geq 90$ GeV the top quark will decay predominantly into 

$$t \rightarrow Wb$$

This provides the following signature for top searches at LHC

$$t\bar{t} \rightarrow W^+bW^-\bar{b} \quad W^+ \rightarrow l\nu$$

or

$$W^- \rightarrow jet \quad jet \quad high \ background$$

$$W^- \rightarrow l\nu \quad low \ background$$

These signatures are illustrated in Fig. 18. Let us first consider the reaction

$$t\bar{t} \rightarrow W\bar{W}bb \rightarrow e\mu + jets$$

which provides the cleaner signature. Requiring two hard ($p \geq 50$ GeV) isolated leptons a signal to background ratio of about 100 can be reached in the mass range of interest (Fig. 19) with rates of the order of $10^4$/y even at a modest integrated luminosity of $10^4pb^{-1}$.

The reaction

$$t\bar{t} \rightarrow W\bar{W}bb \rightarrow l\nu jjb\bar{b}$$

provides a higher rate at the cost of more severe background from $W + jets$ production with the $W$ decaying into $l\nu$. This background can be reduced by stringent cuts on the transverse energy (momentum) of the jets and leptons. Also, the lepton is required to be separated topologically from the rest of the event. This results in a reduction of the signal by about two orders of magnitude, as shown in Fig. 20. The signal to background ratio in the interesting region around $m_t \simeq 150$ GeV is about 2. It could be improved by tagging on the $b$-jets.

In spite of the lower signal to background ratio this reaction may be important for a top mass determination. The suggested procedure is to use the reaction

$$t\bar{t} \rightarrow Wb + \bar{W}b \rightarrow l\nu b + jjj$$

to tag on $l\nu b$ and determine the mass from the other three jets. The result is shown in Fig. 21. Applying a cut of $\delta m_w = \pm 20$ GeV on the invariant mass of two jets (Fig. 21a) allows a top mass determination with a precision of about 8 GeV. Note that only combinatorial background - no background from $W + jets$ - is included in these figures. Using additional $b$-tagging one can hope to improve the precision on the top mass determination to $\delta m_t \simeq 5$ GeV.

4.3 Bottom physics and CP violation

As can be seen in Fig.7 the cross section for $b\bar{b}$ physics is indeed huge at LHC, of the order of $1/10$ mb. Even at $1/10$ of the nominal luminosity this will produce an estimated $10^8$ events/s or $10^{12}B\bar{B}$/year. This opens a large field of beauty physics, in particular rare decays, $B\bar{B}$ oscillations and possibly CP violation.

I will restrict myself to a short discussion of the study of CP violation in the $B^0/\bar{B}^0$ system. If the source of CP violation determined in the $K^0 - \bar{K}^0$ system is a non vanishing phase in the CKM matrix we can try and estimate the magnitude of the effect. In the following we will discuss a possible CP violation in the $B^0/\bar{B}^0$ system in terms of this phase $\eta$. I recall briefly the CKM matrix which can be parametrized in the Wolfenstein notation by $A$, $\lambda$, $\rho$ and $\eta$. 

\[ V_{CKM} = \begin{pmatrix} V_{ud} & V_{us} & V_{ub} \\ V_{cd} & V_{cs} & V_{cb} \\ V_{td} & V_{ts} & V_{tb} \end{pmatrix} = \begin{pmatrix} 1 - \lambda^2/2 & \lambda & A\lambda^3(\rho - i\eta) \\ -\lambda & 1 - \lambda^2/2 & A\lambda^2 \\ A\lambda^3(1 - \rho - i\eta) & -A\lambda^2 & 1 \end{pmatrix} \]

\[ \lambda \approx \sin\Theta_c \approx 0.22 \text{ and } A \approx 1 \text{ are rather well known experimentally, whereas } \rho \text{ and the CP violating phase } \eta \text{ are only poorly known from CP violation in the } K^0/\bar{K}^0 \text{ system, } B^0\bar{B}^0 \text{ mixing and semileptonic decays of the bottom quark (see Fig. 23). Unitarity of the } \]

As the CKM matrix implies the relation

\[ V_{td} + V_{ub}^* \approx \lambda V_{cb} \]

to leading order in \( \lambda \).

The relation can be illustrated in the 'unitarity triangle' of Fig. 23a. To determine \( \eta \) the most sensitive quantity would be the angles \( \phi_1 = \beta \) or \( \phi_3 = \gamma \). The angles could in fact be determined by studying the decay of \( \bar{B}^0 \) into CP eigenstates

\[ B^0 \to f(\text{CP eigenstate}) \]

as indicated in Fig. 23. The best candidate would be the decay

\[ B_d^0 \to J/\psi K^0_s \]

Why this is suited to measure \( \phi_1 \) will become clearer below.

The experimental idea is illustrated in Fig. 22. A pair of \( \bar{b}b \) is produced in pp collisions and fragments into a \( B\bar{B} \) pair, e.g. \( B^0B^- \). The \( B^0(\bar{B}^0) \) can be tagged by its associated \( \bar{B}(B) \). This can either be identified by its decay \( \bar{B} \to l^- + X(B \to l^+ + X) \), by its \( K^\pm \) from the \( b \to c \to s \) cascade, or in the case of charged \( B^\pm \) by complete reconstruction. Only the first method will be fairly simple, the others will need sophisticated trackers with secondary vertex determination or complete reconstruction.

A CP violation would manifest itself in a slight asymmetry in the \( B^0/\bar{B}^0 \) decay. In the example of Fig. 22 this asymmetry could be measured by the (time integrated) asymmetry

\[ A = \frac{N(B^0 \to f)/N(\bar{B}^0 \to f)}{N(B^0) + N(\bar{B}^0)} \]

or, in the case of \( B^0(\bar{B}^0) \to J/\psi K^0_s \) and \( B^0, \bar{B}^0 \) tagging by \( l^-(l^+) \)

\[ A = \frac{N_-(J/\psi K^0_s l^-) - N_+(J/\psi K^0_s l^+)}{N_- + N_+} \]

It can be shown that

\[ A = \frac{x_d}{1 + x_d^2} \sin 2\phi_1 \]

where \( \phi_1 \) is one of the angles in the unitarity triangle (Fig. 23a) under study and the first term with \( x_d = \frac{\Delta M}{\Gamma} \) is the dilution factor due to \( B^0/\bar{B}^0 \) oscillations. The reason why this particular channel measures \( \phi_1 \) is that \( \phi_1 \) is the phase between \( V_{cb} \) which determines the decay in the particular case of \( B_d^0 \to J/\psi K^- \) and \( V_{td} \) which is responsible for the CP violation in \( B_d^0/\bar{B}_d^0 \) oscillations.
Other decay modes could be used to measure the other angles in Fig. 23a, like

\[ B_d^0 \to \pi^+\pi^- \text{ for } \phi_2 \quad B_s^0 \to \rho K_s^0, \pi^+\pi^- \text{ for } \phi_3 \]

however the above mentioned \( B_d^0 \to J/\psi K_s^0 \) has the best signature and avoids problems of \( B_s^0 \to B_d^0 \) confusion which would occur in the \( \pi\pi \) channel or a large dilution in the \( B_s^0 \) case.

Fig. 23b summarizes the present knowledge of \( \rho \) and \( \eta \). It can be estimated that a measurement of \( A \) at LHC could establish CP violation in the \( B^0 \) system with more than 3\( \sigma \) significance if \( \sin 2\phi_1 > 0.15 \).

CP violation experiments have been discussed both for collider and fixed target experiments. In the latter case a gas target could be used or part of the beam would have to be extracted e.g. by a bent crystal. Both experiments have their pro's and con's and none of them will be easy. Much more work has to go into the study of these possibilities before a firm conclusion on its feasibilities can be reached.

4.4 Tau neutrinos

The LHC/SSC may be the first machines where the \( \nu_\tau \) could be directly seen. Tau neutrinos will be produced mainly through c and b decays. The cross section for the dominant source \( \sigma_{D_s} \approx 1 \text{ mb} \) can be estimated up to an uncertainty of about 3. The decay of the \( D_s \)

\[ pp \to D_s^+ \to \nu_\tau \to \tau^+ \to \bar{\nu}_\tau \text{ fast} \]

will produce a fast and a slow component of \( \nu_\tau \)'s. The branching ratio \( D_s^+ \to \tau^+ \nu_\tau \) is theoretically estimated to be of the order of 3 \%. The energy spectrum and angular distribution of the neutrinos is shown in Fig. 24.

<table>
<thead>
<tr>
<th>Table 3: Production rates of ( \nu_\tau ) at LHC, ( 10^7 )s, ( \Delta\Theta = \pm 2.5 \text{ rad} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{\mathcal{L}}(\text{cm}^{-2}\text{s}^{-1}) )</td>
</tr>
<tr>
<td>collider</td>
</tr>
<tr>
<td>gas-jet</td>
</tr>
<tr>
<td>( \rho \sim 4 \times 10^{14} \text{cm}^{-3} )</td>
</tr>
</tbody>
</table>

Table 3 shows the production rates expected at the LHC for different types of experiments. Similar to b-physics experiments could be carried out either in collider mode or fixed target mode where the latter could be either of a gas jet or beam dump type. The Table shows that highest rates could be reached in the collider mode which would have the further advantage that it does not interfere with the other experimental programme.

The \( \nu_\tau \) would be detected by producing \( \tau \) leptons which decay into \( \mu \)'s after a certain decay path. Fig. 25 visualizes a possible \( \nu_\tau \) interaction in a proposed detector of fibre bundles. The signature for the reaction

\[ \nu_\tau \to \tau X \to \mu \nu_\mu \nu_\tau X \]

would be a non zero impact parameter of the \( \mu \) with respect to the \( \tau \) production vertex which can be reconstructed from the particles produced at the \( \nu_\tau \) - \( \tau \) vertex. The background estimate is less than 10 \% for transverse momenta of \( p_\perp(\mu) > 0.2 \text{ GeV} \). The background is mainly due to c, b decays.

As seen in Fig. 25 a possible \( \nu_\tau \) detector could be placed downstream an LHC interaction point and could easily parasite to any LHC experiment. The rates of Table 3 have been calculated for a 10 m long detector of 2 kg/cm\(^2\) density which would be placed at 100 m from the interaction point.
4.5 Physics beyond the standard model

I will be very brief on this subject which has been excellently and comprehensively summarized on the LHC workshop by F. Pauss [4].

Ideas on how physics beyond the Standard Model may look like cluster around the subjects of supersymmetry, new vector bosons and alternative symmetry breaking schemes. The task of the three corresponding working groups was to find out whether or not these types of 'predicted' new physics phenomena can be detected at LHC.

In a nutshell the answer is given in Fig. 26. It compiles a few typical examples for new phenomena like supersymmetric gluons \( \tilde{g} \), new vector bosons \( Z' \) and \( W_R^+ \), and an example of alternative symmetry breaking (Bess-Model). The ratio of expected cross sections to the total cross section is given as a function of the new particles' masses. As can be seen, cross sections are comparable to the hard but doable \( H^0 \rightarrow 4\mu \) channel at masses of the order of 1 TeV. The general conclusion of extensive studies is the same: the LHC discovery potential for new phenomena in the TeV range extends over most of the predicted phenomena.

5 Some Remarks on LHC Logistics

Fig. 27 shows the principle layout of the interaction areas at LHC and LEP. The pits 1 and 7 will be prepared for LHC experiments whereas pit 5 and 3 will house the LHC dump and scrapers for beam cleaning respectively. The four LEP experiments are supposed to stay in their position. In principle however, also these areas could be used for LHC experiments.

The present time planning for LHC construction is shown in Fig.28. It assumes installation of SC cavities into LEP during 4 winter shutdowns in 91 to 94 which would allow for three years of LEP 200 operation at maximum energy. With 192 cavities installed this maximum beam energy will be somewhere around 90 GeV depending on cavity performance.

After LHC commissioning the time planning foresees alternating operation of both LEP and LHC. Personally I am convinced that this will not happen in the first two or three years of LHC operation. It may well be considered later depending on the way our knowledge of physics will evolve. After a final installation of the LHC machine and detectors commissioning of the machine is aimed at in 1988.

6 Summary

The LHC opens up a regime where to the best of our knowledge new physics will show up.

'Standard' Higgs \( H^0 \):

If it exists, the LHC will be able to see \( H^0 \) in the mass range \( \geq 130 \text{ GeV} \) and with proper instrumentation (\( H \rightarrow \gamma\gamma \)) also in the 'gap' 80 - 130 GeV.

\textbf{T}OP physics:

LHC will be a heavy quark factory. Top will be produced in large quantities. A mass determination with \( \delta m_t \leq 5 \text{ GeV} \) may be feasible.

\textbf{B}\textsuperscript{0} physics and CP violation:

\( b\bar{b} \) will be produced at high rates exceeding 'B-factories'. Very ambitious instrumentation will however be needed to make use of it, e.g. for the study of CP violation.

\( \tau \) physics:

LHC may offer the first chance to detect tau neutrinos.

\textbf{Beyond the 'standard model'}:

The door will be open to a new energy regime!
REFERENCES

[1] Proceedings of the Large Hadron Collider Workshop,
    Aachen, 4-9. Oct. 1990, CERN 90-10
[5] Proceedings of the Workshop on Physics at Future Accelerators,
    La Thuile, 7-13 Jan 1987, CERN 87-07.

List of Figures
1 Consistency limits for the standard model as a function of the cutoff $\Lambda$.  10
2 The CERN machine complex including LHC.  10
3 LHC in the LEP tunnel.  11
4 Mockup of the LHC in the LEP tunnel.  11
5 Cross section of an LHC dipole magnet.  12
6 Full length 'two-in-one' prototype magnet (HERA coils).  12
7 pp cross sections as a function of energy.  13
8 Standard Higgs production and cross sections.  14
9 Higgs decay width as a function of Higgs mass.  14
10 Cross section and rates for the process $pp \rightarrow Z^0 Z^0$ and $Z^0 Z^*$ with decay $Z^0$, $Z^* \rightarrow 2 \mu$.  15
11 Invariant mass distribution of $H \rightarrow 4l$ for two different Higgs masses.  16
12 Invariant mass distribution of $H \rightarrow 4l$ with background (c.f. text for details).  17
13 Invariant mass of $H \rightarrow 4l$ for masses below $2m_Z$.  18
14 a) Cross section times branching ratio for $pp \rightarrow \gamma \gamma$. b) Signal to background for $H \rightarrow \gamma \gamma$.  19
15 $H \rightarrow \gamma \gamma$ in W bremsstrahlung.  20
16 Higgs mass range covered by various signatures.  20
17 Top cross section as a function of top mass.  21
18 Signatures for top search.  21
19 The reaction $t\bar{t} \rightarrow W\bar{W}\, b\bar{b} \rightarrow e\mu + jets$ compared to the background $b\bar{b} \rightarrow e\mu + X$.  22
20 Cross section $t\bar{t} \rightarrow l\nu + jets$ (c.f. Fig. 19).  22
21 Invariant mass distributions for 2 and 3 jets after tagging on $t \rightarrow Wb \rightarrow l\nu b$.  23
22 Searching for CP violation in $B^0/\bar{B}^0$.  23
23 The unitarity triangle and the experimental knowledge of $\rho$ and $\eta$.  24
24 Energy spectrum and angular distribution of $\tau$ neutrinos from $D_s$ decay.  25
25 A possible $\nu_\tau$ detector at LHC and the signature for $\nu_\tau$.  26
26 Some predictions of cross sections for physics beyond the standard model.  27
27 Layout of the experimental areas at LEP/LHC.  27
28 Time planning for the LHC.  28
Figure 1: Consistency limits for the standard model as a function of the cutoff $\Lambda$.

Figure 2: The CERN machine complex including LHC.
Figure 3: LHC in the LEP tunnel.

Figure 4: Mockup of the LHC in the LEP tunnel.
Figure 5: Cross section of an LHC dipole magnet.

Figure 6: Full length 'two-in-one' prototype magnet (HERA coils).
Figure 7: pp cross sections as a function of energy.
Figure 8: Standard Higgs production and cross sections.

Figure 9: Higgs decay width as a function of Higgs mass.
Figure 10: Cross section and rates for the process $pp \rightarrow Z^0Z^0$ and $Z^0Z^{\pm}$ with decay $Z^0$, $Z^0 \rightarrow 2\mu$. 
Figure 11: Invariant mass distribution of $H \rightarrow 4l$ for two different Higgs masses.
Figure 12: Invariant mass distribution of $H \rightarrow 4\ell$ with background (c.f. text for details).
Figure 13: Invariant mass of $H \rightarrow 4l$ for masses below $2m_Z$. 
Figure 14: a) Cross section times branching ratio for $pp \rightarrow \gamma\gamma$. b) Signal to background for $H \rightarrow \gamma\gamma$. 
Figure 15: $H \rightarrow \gamma \gamma$ in W bremsstrahlung.

SEARCHING FOR THE HIGGS AT LHC

- $(H?) \rightarrow 2Z \rightarrow 4$ jets
- $H \rightarrow 2Z \rightarrow 2\nu + 2\mu$
- $H \rightarrow 2Z \rightarrow 4$ leptons ($\mu$, $e$)
- $H \rightarrow Z + Z^* \rightarrow$ leptons
- $H \rightarrow Z \gamma \gamma$

Figure 16: Higgs mass range covered by various signatures.
Figure 17: Top cross section as a function of top mass.

Figure 18: Signatures for top search.
Figure 19: The reaction $t\bar{t} \rightarrow W\bar{W} b\bar{b} \rightarrow e\mu + \text{jets}$ compared to the background $b\bar{b} \rightarrow e\mu + X$.

The lepton isolation requires a total $p < 10$ GeV from other particles within a cone of $\delta R = 0.4$ around the lepton, where $\delta R^2 = \delta y^2 + \delta \phi^2$, $y =$ rapidity, $\phi =$ azimuthal angle.

Figure 20: Cross section $t\bar{t} \rightarrow l\nu + \text{jets}$ (c.f. Fig. 19).
Figure 21: Invariant mass distributions for 2 and 3 jets after tagging on $t \rightarrow Wb \rightarrow l\nu b$.

Figure 22: Searching for CP violation in $B^0/\bar{B}^0$. 
Figure 23: The unitarity triangle and the experimental knowledge of $\rho$ and $\eta$. 
Figure 24: Energy spectrum and angular distribution of $\tau$ neutrinos from $D_s$ decay.
Figure 25: A possible $\nu_r$ detector at LHC and the signature for $\nu_r$. 
Figure 26: Some predictions of cross sections for physics beyond the standard model.

Figure 27: Layout of the experimental areas at LEP/LHC.
TIME SCALE FOR THE CONSTRUCTION OF THE LHC IN THE LEP TUNNEL

Figure 28: Time planning for the LHC.
What Have We Learned from LEP and HERA?

C. N. P. Gee
Rutherford Appleton Laboratory, UK

Abstract

It is nearly two years since computer systems in the four LEP experiments collected and processed the first real events. The two HERA experiments are expected to observe colliding beams before the year ends. These six large experiments have been faced with a profusion of new computing options in hardware (wide availability of workstations, new buses, microprocessors) and software (knowledge based systems, object-oriented languages, simulation & modelling tools and more) plus plenty of time to think about them.

In these lectures I assess the use made of the new tools and the way in which the groups have worked, highlighting their experiences and the most recent developments underway. The objective is not to describe any of the tools in detail, but to identify how and why they have given good or bad results.

1 Historical Background

It is instructive to start this review of LEP and HERA experiences with a brief look at the environment in which the software for LEP experiments was designed. Among experiments then prominent were JADE and TASSO at the PETRA accelerator (DESY), the CDF experiment at Fermilab, and OMEGA and UA1 at Cern.

![TASSO CAMAC CONFIGURATION](image)

Figure 1: The Camac configuration in the TASSO experiment at DESY in 1981

Tasso[1] and Jade[2] both used Nord computers connected to large Camac systems - Tasso for example with 5 branches and 24 crates (figure 1).

The UA1 experiment[3] ran with a large Camac system (200 crates), events being managed by around 30 processors in VME crates controlled by Macintosh PCs. A Generation of Cern experiments starting with WA42[4] used Vax computers with Camac and
Camac variants such as Romulus. The CDF experiment at FNAL was among the first to design a system using Fastbus.

All the LEP experiments chose Vax systems as their main experiment computers, although the then current Nord systems were some 30 percent faster. The choice emphasised the need for good hardware and software support and a versatile operating system. Certainly, use of the Vax/VMS operating system seems to have been highly beneficial in the experiments where it has been fully exploited.

To read data from the experiment, the LEP and HERA experiments rely on a mixture of VME, Camac, Fastbus and detector-specific front ends, connected to a standard central readout system. For the latter, Aleph, Delphi, and L3 all use Fastbus, while Opal, H1, and Zeus are based on VME.

2 Use of Fastbus

The developers and users of Fastbus hoped that it might be widely adopted in industry. This did not happen, perhaps because the speed of Camac was adequate for industrial use. Nevertheless, Fastbus has been used successfully by many experiments (with more than 500 crates in use at Cern by March 1990[5]). User groups have managed to build their own modules, and a reasonable range of commercial modules are available. The main difficulty in hardware seems to be the high cost of power supplies, and in software substantial effort has been needed to make best use of the Fastbus routing options.

Fastbus is optimised for movement of data, and users routinely achieve 60 Mbytes/sec in block transfers. This rate can be boosted to around 200 Mbytes in crates with ecl backplanes.

The Fastbus systems used at LEP are hosted by Vax computers (Figure 4), for which substantial software development has been needed.

3 Evolution of Online System Software

When online software development for LEP was starting, a closely integrated data acquisition suite was in widespread use on Cern Vax systems, having been adapted from the PDP-11 system run at EMC[6]. The heart of this system was a complex data management program which controlled all data flow to and from a large event buffer.

![Diagram showing strategies for accessing events in a buffer](image)

Figure 2: Strategies for accessing events in a buffer. On the left, a view of the older "DAQP" strategy. On the right, the new MBM strategy, in which each client program includes the access and arbitration routines.

It was established in early detector tests for LEP that this system had difficulty with large events. A particular problem was the transmission of events to monitoring programmes. Meanwhile, the CDF group at FNAL had developed new Vax software similar
in concept to that used in the Tasso data acquisition system, and this was adopted by Aleph. The Cern-DD Online group elected to develop a new modular data acquisition system[7], including a new buffer manager (MBM)[8] which has been widely used. The essential difference in the new software was that the buffer arbitration and access routines were linked in as part of all client programs (figure 2). The decision in this software to make active use of memory mapping features has had an important influence in many areas of online systems.

![Diagram of buffer mapping](image)

Figure 3: Buffer Mapping in an online environment. Tasks 1 and 2 map the complete event buffer, but to different virtual addresses. Task 3 maps a single event, and task 4 maps only part of an event. All these views should be valid.

4 Online Memory Management using Zebra and Bos

The Zebra memory manager[9] was developed at the same time as the LEP experiments, and has found wide acceptance at LEP, HERA and elsewhere (except in Aleph, where BOS[10] is used). However, several problems have been met with use of memory managers for data formatting, which the experiments need to monitor events part way through the acquisition process.

It seems useful to note here some of the problems encountered. All the LEP experiments have written their own private Zebra or Bos versions (more than one, in some cases!) to overcome these (figure 3).

- Memory Allocation. Contiguous space is allocated by the event buffer manager, and can take into account the size to which an event will grow, perhaps as a function of the trigger type. Memory manager banks must be created within this space.
- Concurrent Allocation. In a parallel environment, more than one program may need to create and fill banks in an event.
- Mapped Concurrent Access. Many programs can be connected to one event (or to parts of an event). Are they all granted valid views?
- Garbage Collection. The event buffer manager controls event buffering based on it's knowledge of event sizes and consumers. Correct sizing of the event buffer means that garbage collection is never needed.
- Error and Diagnostic Messages. Buffer managers should be able to use the central message processing services provided within Online systems, rather than just using Fortran "Write" statements.
Multiple Events. A buffer manager may contain many events with the same bank names in various stages of construction.

5 VME and Microprocessors

The VMEbus was designed to support microprocessors. With the arrival of the Motorola 68000 family of microprocessors, its use in industry has become widespread, with a range of processor and memory modules commercially available. VMEbus is slower than Fastbus - although the bus is rated at 20 Mbytes/sec, it is in practice hard to achieve more than around 6 Mbyte/sec in a DMA transfer. Developments are underway to raise the theoretical bus speed to around 80 Mbytes/sec.

Lacking a standard crate interconnect (although this is now changing), groups have built their own. H1 have developed a fibre-optic ring capable of sustained data rates of 12.5 Mbyte/sec[11], while Zeus use transputer links with crossbar switches rated at 26 Mbytes/sec[11]. Opal use a ribbon cable link to daisy-chain crates (figure 4).

Figure 4: (a) The Aleph event readout, using 120 Fastbus crates, and (b) part of the OPAL readout using VME. Both experiments are controlled by Vax cluster systems.

5.1 VME Microprocessor Software

H1 (like UA1) run a system with microprocessors closely linked to Macintosh PCs. The microprocessors use no formal operating system, relying on the PCs for file handling, etc.

Opal and Zeus both use the OS9 operating system[12], although in Zeus OS9 is not used in real-time applications. In OPAL, microprocessor software development was started without an operating system, and the change of direction to use OS9 resulted in abandonment of a substantial body of working code. Early developments under OS9 were extremely difficult owing to unreliability of the system and inadequate networking, coupled with the explosive growth in demand for memory (in the running version of this system, a typical processor needs more than 150 memory-resident code modules to be loaded at boot time).
When embedding processors into a readout system, it is important that processing capacity is positioned optimally in the readout around the points where trigger decisions are applied. For example, there is no point compressing data from every event if most events are rejected, unless compressed data is needed for the trigger decision itself. At LEP this issue may become more important if background rates rise with 8*8 bunch operation.

A VME system may consist of modules from several manufacturers, and experience shows that not all combinations work together. A dedicated team of hardware and system software experts is essential to isolate problems.

6 Experience with Ethernet

Good network communications, including at least file transfer, remote login and a reliable network transport service are essential when using many microprocessors.

All the experiments at LEP and HERA are using Ethernet for control messages and general communications. Thin Ethernet is much preferred because of the cable flexibility, lower cost, ease of connection and elimination of the full specification transceivers and interface (AUI) cables. Experience shows that Ethernet repeaters with automatic disconnection and indication of faulty segments are essential, and LAN traffic monitors are useful. It is also important to design the network topology such that traffic can be partitioned by bridges as the load increases - an area where several groups are beginning to have difficulties.

For some reason, it is difficult to convince members of experimental teams of the need to maintain continuity of Ethernet segments when disconnecting equipment (This is why the automatic repeaters are essential!), and the time when the experiment is installed is particularly fraught. Both Aleph and Delphi have used thin Ethernet cabling with crimped tee pieces to minimise this problem.

Once stable, few Ethernet faults are observed. Full specification transceivers in the (metal) OPAL computer room were damaged during an electrical storm in 1988. The Ethernet segment involved connected this room to nearby permanent buildings. At the experimental site, a fibre optic link has been used between the surface and underground installations, although other experiments with copper ethernet links have reported no problems.

7 Partitioning and Detector Monitoring

The term “Partitioning” was first used by Aleph. It describes the ability of a data acquisition suite to function as several independent units (Partitions), each controlling and reading data from a different part of the experiment. In its most general form, partitioning allows a separate experiment with data processing and recording to be run for each detector component.

Partitioning needs to be considered early in the design of experiment support software, for example to enable routing of error messages to an appropriate partition-specific point. Requirements may also be imposed on hardware - for example in the design of a trigger system capable of interleaving individual detector triggers. But although the resulting system is substantially more complex, partitioning has been found extremely valuable for detector testing and at the critical time when the experiment is being assembled. The Vax-based experiments report that the VMS cluster features, including the distributed lock manager, have been valuable. Agreement to use a single type of computer for all detectors is clearly also important.

In a system with many levels of triggering and processing, monitoring may be needed at several different points. Some groups send event samples to a common point to accomplish this. But care needs to be taken in interpreting monitoring results if they are accumulated from an impure event sample early in the readout tree.
8 Controlling the System

The rapid growth in the number of computers involved in data acquisition has led to a demand for unified control systems, and all six experiments have software providing a single control point under normal operating conditions. They cover detector operating conditions ("slow controls"), trigger configuration, links with the accelerator computers, control and monitoring of the event data flow, and possibly event reconstruction.

Although differing in detail, these systems all rely on the concept of programs with well-defined state models communicating with a central controller from which commands are issued. The Delphi and L3 groups are using the State Manager Interface (SMI) software\textsuperscript{[7]} developed at Cern, while Aleph, OPAL, H1 and Zeus have all developed their own control systems. Most groups have also made an attempt to use a standard interactive style for all online programs, although many offline programs have not been made to conform.

Regarding slow controls, the message from all the experiments is to make absolutely all detector operations possible under software control - there is no benefit to be gained in installing reset buttons requiring manual intervention in remote parts of an experiment.

9 Xwindows

X-windows provides the capability for one computer to create workstation displays locally and on a remote computer screen. In online systems, this functionality is in constant demand, for example for histogram display. In common with other windowing system applications, code needs careful design:

- X-windows is complex and needs expertise to program. Applications must be prepared to process in an order determined by the user or the window manager (mouse motion, hiding and exposure of windows) rather than in a predetermined sequential manner.
- GKS graphics drivers have (so far) only limited X functionality

![Figure 5: An example of an X-windows application - an interactive panel forming part of the OPAL Run Control suite](image)

- The supporting code (at least on Vax systems) is large.
- The application has to detect and correct network errors.
- Slow updating and high network occupancy can result if screen updates are requested for minor display changed - for example, updating a displayed scatter plot as each
bin is incremented. But good X-windows code produces fast, high-quality displays, out-performing other screen managers.

- For some applications, it may be preferable to split the application and run one part at the displaying workstation.

X-Windows is emerging as a common windowing standard available on Unix and Non-Unix systems, so there is for the first time the prospect of using standard tools with a wide user base to create high-quality portable interactive displays (Figure 5).

10 Computer Aided Engineering

When LEP and HERA experiments were being designed, it was usual for engineering designs to be moved between institutes on paper (as blueprints). Component dimensions were often read from drawings and entered by hand into CAE systems, even in the same institute if it was necessary to use more than one CAE system (Figure 6). Later, the same information has again been entered manually into the detector geometry databases used by MonteCarlo and event reconstruction programs.

![Figure 6: Detail of an OPAL detector component whose dimensions were transferred between programs by hand](image)

International standard formats now exist for exchange of engineering drawings, and a project is underway to enable experiment designs to be exchanged between CAE systems at different institutes[19]. Several exchange formats exist, among which IGES, the Initial Graphics Exchange Specification, and SET, the Standard d'Échange et de Transfer are prominent. A proposed standard exchange format, STEP, is under development, but several years may elapse before conformant CAE systems are available. Interfaces between CAE systems and these formats are usually incomplete, and files produced by one system are frequently unreadable on another. Tests have established that easy design interchange is some way off. An attempt is also being made to connect the Geant MonteCarlo program to these formats, but creation of the required bi-directional link appears complex.

11 Unix and POSIX in Experiments

The entrance of Unix into the world of HEP is undeniably connected to the arrival of RISC processors - Unix is used just because it gives access to fast and cheap computing capacity and graphics.
Any user with experience on many systems will have his own feelings about Unix. Mine include ...

**Good Features**
- Speed, Cost
- Multi-Vendor
- Fortran and C Available
- Lots of Freebies
- Make - a good building tool
- Background jobs via & etc

**Bad Features**
- Obscure Command Names
- Poor Documentation
- Weak System Management Tools
- No Batch system
- No memory use controls
- No Realtime features

The window system and associated utilities (including editors) need learning afresh on every system, the file structure is strange, and it's hard to find out what the system is doing with it's resources. However, these comments are also true of many other operating systems.

Unix seems to be good for compute-intensive work, particularly if the application has a good user interface, and the IEEE arithmetic offered by some processors is useful on experiments that have invested in M68000 family processors. However, application development and debugging are easier using other systems.

The range of applications that can be moved to Unix is likely to increase dramatically as POSIX (Portable Operating System Interface)[13] gains ground. POSIX draft 1003.4 - "Realtime Extension for Portable Operating Systems"[14] - aims to define standard interfaces for a range of realtime services, including

- Binary Semaphores
- Shared Memory and Memory-Mapped files
- Realtime signals with Guaranteed delivery
- Interprocess message passing
- Fast Contiguous file operations
- Mutexes (resource locking)

- Memory locking
- Pre-emptive Priority Scheduling
- Clocks and Timers
- Synchronous and Asynchronous I/O
- Multiply threaded processes

Several manufacturers are already committed to parts of POSIX, including some non-Unix vendors. Meanwhile, some Cern online programs are being converted to run inside a Virtual Operating System VOS. The success of POSIX and VOS should be of enormous benefit to future experiments.

12 Use of Databases

The response by LEP and HERA experiments to database opportunities has been varied. All are using databases of some sort. Groups that have tried relational databases are convinced of their usefulness for some work. Sadly, others are firmly convinced that relational databases have nothing to offer, so have not tried them. So substantial ground remains to be covered before databases are properly integrated into experiments, and ignorance is a major problem.

Many experiment database applications can be identified[15]:

- Address Lists
- Experiment book-keeping
- Detector Geometry
- Event Data
- Histograms
- Publications
- Fastbus System Management

- Email Addresses
- Online Databases
- Calibration Constants
- Program Versions
- Software Documentation
- Detector Hardware Inventory
- Tape and Run Production

Oracle[16] is probably the most popular relational database at present used in experiments. This tool works across networks, is robust, and comes with a range of interactive and display tools. But despite the special European terms negotiated by Cern with the supplier, it has not been installed in all institutes.
Figure 7: The Aleph event reconstruction facility. Disks are filled with events by the online system and then remounted for use by the analysis farm. No data format conversions are needed.

Programming relational database queries is tricky in record-oriented languages such as Fortran, since the general response to a query is a table rather than a record, and the number of records in the table is indeterminate. Object-oriented programming is likely to help in this area.

Calibration databases are a special case where use of a dedicated database can perhaps be justified. Calibration results are invariably used as a group, and updates frequently change only a few values.

13 Event Reconstruction and MonteCarlo

With the advent of powerful and cheap Unix processors, it has become possible, and indeed desirable, to reconstruct events close to the experiment. With good organisation, adequate CPU capacity and stable calibration constants, online reconstruction of events in this way processes the events through the longest analysis stage with minimal human intervention.

For total success, the reconstruction scheme has to be seen as part of the online system from the start. This has far reaching implications for reconstruction program design. The Aleph scheme (Figure 7) [17] is elegant, although similar versions are running elsewhere.

Unix workstations are also well suited for MonteCarlo event generation, and various projects are underway. Schemes for almost limitless MonteCarlo event generation are on the horizon. Faced with such facilities, collaborations must nevertheless ensure that their simulation programs remain portable across many cpu type, so that generation of small MonteCarlo samples by users in their institutes remains viable.

14 Strange Computers Old and New

Many tales can be told of different attempts to build cheap computers for Particle Physics applications.
At the start of the LEP construction, the Canadian Government was funding a project to build Vax emulators. This was abandoned when it was found that emulating the (extensive) Vax instruction set would not be cost effective.

At the same time, Delphi had acquired several 370/E emulators. These were given to OPAL, where they were made to work but then retired without being used. In Delphi, 3081/E emulators at level 3 and 4 in the readout were also commissioned, but never used. However, 3081/E processors are working in L3.

In a joint project with Digital, Aleph built and are using a farm of MicroVax processors on the BI bus of their main Data Acquisition Vax computer. H1 will use a farm of R3000 (Risc) microprocessors in a Farm. Zeus planned a similar farm, but have recently decided instead to use a farm of Silicon Graphics workstations.

The moral from this tale is that, while it was once economic to build special-purpose computers like the PDP-11/E, 360/E, 370/E and 3081/E, it no longer is. It may (for the present) still be interesting to buy cpu chips and assemble fast processing cards, but this still leaves substantial hardware and system software to be developed. An easier route today is to work with industry, for example by negotiating economical terms for purchase of workstations without screens or in joint development projects.

![Diagram](image)

Figure 8: The AMT DAP 510 with 1024 processors in a SIMD structure. This class of machine excels for matrix problems, but seems less suitable for particle physics programs

Regarding the different computer architectures met over the last few years, three classes can be distinguished:

- SIMD Machines, such as the AMT DAP(Figure 8)[18] and the Connection Machine. It seems difficult to adapt HEP code to run on these architectures.
- Fast scalar machines with vector instructions, such as the current generation of Cray and IBM processors. Vectorising has given limited speed increases in a few cases, but generally at the cost of substantial effort. The resulting programs usually contain non-portable compiler directives. Better compilers would certainly help, but many particle physics problems seem intrinsically unsuited to this architecture also.
- MIMD systems, including farms of workstations. These are a great success. Complete
systems have to be assembled at each site, however, and careful setup is needed to avoid I/O bottlenecks. Happily, the computer industry is interested in pursuing R & D on such architectures.

15 International Working

International networks are becoming faster and more robust. They are also expensive, particularly within Europe. Applications rapidly expand to fill available capacity, so it is not likely that international interactive working will become popular.

De-facto standards frequently turn out to be more useful than internationally agreed ones - compare, for example, the popularity of Exabyte cartridges, TCP/IP and DECNET networking, and \TeX word processing with the use of standards such as GKS-3D graphics, OSI network protocols, and SGML text processing.

Given that parts of any experiment team have to work away from the experiment site, proper organisation is crucial to make best use of their time. This implies that formal mechanisms to maintain contact should be designed and used from the start of the experiment. The experiments could gain considerably by sharing and using tools for distribution of software, documentation, papers, data tapes, minutes etc.

A notable success in this area is the HEPVM collaboration, which is highly valuable in ensuring a common working environment between institutes with IBM systems. A similar organisation is probably needed to avoid anarchy among Unix systems.

16 Human Nature

Computing for a large experiment is inherently hard to organise. It relies on contributions from collaborators from many different institutes, there is no single point of executive control, and a willingness to conform is essential. In discussions for these lectures with the experiments, several cases have come to light in which coordinators were seemingly unable to prevent several institutes or individuals within their own collaborations from working in competition to develop the same hardware or software. Regrettably little software has been exchanged between collaborations, and this is perhaps an area in which the senior committees should take a greater interest. If the efforts wasted in software (and hardware) duplication can be instead used to enhance existing products, the most important lesson of all will have been learned.

17 A Summary - What have we learned?
- A computer with a decent operating system is superior to a slightly faster one without.
- A standard detector computer is essential for testing, monitoring and control.
- Software development and detector testing on a microprocessor with an immature operating system should be avoided.
- A good control system covering all aspects of the experiment is both feasible and worthwhile.
- Ethernet is popular. It’s reliable if you can prevent your colleagues from disconnecting it.
- X-windows works well, but is big and needs expertise to program. Some applications need to be split to achieve reasonable display speed.
- Slow controls should be arranged so that no human intervention is ever needed - particularly in distant parts of the experiment.
- It may eventually be possible to transfer engineering design information between CAE, MonteCarlo and event reconstruction programs.
- Databases work. Use standard tools where possible. Substantial training is needed in the community.
- There are mixed feelings about Unix. POSIX may become crucial in design of portable time-critical code.
- Farms probably offer the best route to high-volume HEP computing. Other architectures are less useful.
- Standards for all experiment computing should be laid down together. Programs, libraries and utilities traditionally regarded as “offline” may well have to run in the online environment.
- Careful organisation is needed from the beginning to ensure that best use is made of people and computers remote from the experiment (i.e in different countries).
- Experiment coordinators and the high level review committees should work together to reduce duplication of software between experiments.

REFERENCES

[1] D R Quarrie, The Data Acquisition System for the TASSO Experiment, RL-81-048
[7] Cern ECP Division Data Acquisition Systems group, MODEL Modular Data Acquisition System
[9] R. Brun et al, Zebra Dynamic Data Structure and Memory Manager, Cern Program Library Q100
[10] V Blobel, BOS: bank organisation system - dynamic storage organisation with FORTRAN, DESYF 147701
[12] The OS9 operating system, Microware Systems Corporation,1900 N.W. 114th Street, Des Moines, IOWA 50322.
[15] For example, see Computing at Cern in the 1990’s, Chapter 12, July 1989
[16] ORACLE Relational Database. Oracle Corporation, 20 Davis Drive, Belmont, California 94002, USA.
[18] Active Memory Technology Ltd, 65 Suttons Park Avenue, Reading, RG6 1AZ, UK
[19] For an introduction to this subject, see LHC News No 1 (Supplement to Cern Courier), July 1991.
The physics at the LHC and its impact on the experiments

John Renner Hansen/Niels Bohr Institute

November 22, 1991

Abstract

The new proton-proton collider facility in Europe (LHC) will challenge the art of detector design, and it will push the state of the art of electronics and software far beyond its present level. Short time intervals between bunch crossings and many overlapping events make it interesting to handle without losing the physics signal. This paper discusses some of the physics arguments, which lead to the decision of having high luminosity and short time between bunch crossings. Some of the problems related to the decision and possible solutions will be discussed.

1 The physics to study at a future collider.

Despite of the success of the high energy physics accelerators and experiments, many questions about the smallest building blocks in nature and the forces acting between them are still waiting to be answered. All observations made so far are well described within the particle physics Standard Model, the Glashow-Weinberg-Salam model [1]. Unfortunately this model itself contains more open questions, than it answers. The Standard Model operates with three types of the elementary particles as shown in Table 1. First we have the building blocks of ordinary matter, the fermions. It is characteristic for them, that two identical fermions can not occupy the same part of phase space. That is why matter takes up room. Fermions are divided into three families of four particles each. Eleven out of the twelve particles are observed in nature, but the most heavy of them, the top-quark, are still not discovered.

Next we have the mediators of forces. In contrast to the fermions they can occupy the same part of phase space and by that create large macroscopic fields. The photon is the mediator of the electromagnetic force. The $W^+$, $W^-$ and $Z^0$ mediate the weak nuclear force and the gluons act as the mediators of the strong nuclear force. As usual we neglect gravity. The Standard Model prediction of the existence of the $Z^0$ particle, was one of the early successes of the model. The first sign of the $Z^0$ was observed at CERN in 1973 by the Gargamelle neutrino experiment [2], but it was only ten years later, in 1983, that it could be produced as a real particle in proton-antiproton collisions at the CERN SppS collider[3].

The third group of particles has in the simplest form of the model only one member. It is called the Higgs-particle. Its sole purpose is to give masses to the other particles through its interaction with them. It is in the model described by a special theoretical construction, the Higgs mechanism. The Higgs-particle has not yet been observed in nature. It might be that the Higgs-particle does not exist, but then another effect must be observed, which is able to explain, that some particles have a mass.

As many as 27 parameters are not predicted by the model, but must be entered by hand from experimental results. The actual values of the quark and lepton masses and the coupling constants are examples. Most theorists believe, that sooner or later a general theory must be developed to describe nature in a more natural way, than the Standard Model does. In particular the generation of masses. More experimental input is now needed to set the direction of the future theoretical development.

The most precise instrument to test the Standard Model has so far been LEP with its four general purpose experiments, ALEPH, DELPHI, OPAL and L3. Beside its ability to make precise
### Fermions.
*(Spin-\(\frac{1}{2}\) particles)*

<table>
<thead>
<tr>
<th>Quarks</th>
<th>Leptons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electrical charge</td>
<td>- (\frac{1}{3})</td>
</tr>
<tr>
<td>1. family</td>
<td>Down</td>
</tr>
<tr>
<td>2. family</td>
<td>Strange</td>
</tr>
<tr>
<td>3. family</td>
<td>Bottom</td>
</tr>
</tbody>
</table>

### Bosons
*Spin-1 particles*

<table>
<thead>
<tr>
<th>Electrical charge</th>
<th>W(^+)</th>
<th>W(^-)</th>
<th>Z(^0)</th>
<th>Gamma ((\gamma))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electroweak force</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strong force</td>
<td></td>
<td></td>
<td></td>
<td>Gluon ((g))</td>
</tr>
</tbody>
</table>

### Higgs particle(s)
*Spin-0 particle(s)*

<table>
<thead>
<tr>
<th>Electrical charge</th>
<th>H(^0)</th>
<th>?</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mass generator</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Elementary building blocks of matter and forces.
tests of the Standard Model predictions, LEP has also been a unique test bench for new theoretical constructions, which, if correct, would bring our understanding further than the Standard Model. However, none of the many different theoretical extensions to the Standard Model, such as Super Symmetry, Compositeness or Technicolor, have either been confirmed or been ruled out by this common effort. Nor have the two important parameters, the mass of the top-quark and the mass of the Higgs particle so far been precisely determined. From the LEP data we now know, that the mass of the Higgs-particle must be higher than 58 GeV \(^1\), and theory tells us, that its mass must be smaller than 1000 GeV. By collecting all the knowledge we have from many different experiments around the world and assuming the validity of the Standard Model, the mass interval in which we must find the top-quark has been reduced to between 90 GeV and 200 GeV.

To test the Standard Model even harder and to find solutions to some or all of these open questions, it has been proposed to build a new proton-proton collider at CERN.

2 Particle Colliders for high energy physics experiments.

The high energy proton-antiproton colliders at CERN and FNAL and the electron-positron colliders SLC at SLAC and LEP at CERN were designed to make discoveries and explore the physics in the mass region around the heavy intermediate vector bosons W and Z. When the CERN proton-antiproton collider was designed, the \(Z^0\) and W particles were not yet found. The prediction for the masses were roughly 90 GeV for the \(Z^0\) and 80 GeV for the W's \(^2\). To make a resonable rate of collisions between two quarks with a center of mass energy of 90 GeV, one needs collisions between protons and antiprotons with laboratory energies of roughly 300 GeV each. This is because the proton and the antiproton are not fundamental particles, but they have subconstituents inside, the quarks and gluons. Each of these partons carries a fraction of the proton or anti-proton energy. The proton-antiproton collider at CERN started out with an beam particle energy of 273 GeV and was later upgraded to 315 GeV.

When the proton energy is further increased, the energy is distributed on more partons, a phenomena which is called scale braking. At the same time are the production cross-sections for heavy particles smaller than for light particles, so if one wants, in a reasonably short time, to explore a new mass region up to 1000 GeV with a proton-proton collider, beams with particle energy of roughly 8000 GeV are needed. The energy in the center of mass of the colliding protons must be more than 16 times higher than the mass of the particle one wants to create.

One problem with the proton-proton collisions is, that more than 85% of the energy is not involved in the hard scattering process, but it is contaminating the event with spurious signals uncorrelated to the interesting physics process.

At an electron-positron collider, for instance LEP, the colliding particles are fundamental. When the electron and the positron are given an energy of 50 GeV each, 100 GeV is available for the production of new particles. Another advantage of an electron-positron collider over a hadron collider is, that no remaining energy is disturbing the event signature. A natural question is then to ask; why not let the future particle physics research be based on the much cleaner e\(^+\)e\(^-\) collisions, which so successful have been used at LEP? The experiments will be much simpler. With minor modifications one of the existing LEP detectors could be used. But the problems are moved from the detectors to the machine. The beam intensity requires special attention. Calculations show that, to get sufficiently many interesting events with a beam energy of 1 TeV it is necessary to squeeze the beams to a transverse size of only 65 nm, before they collide \([4]\). This is with the present technology not possible, and it will most likely not be possible before the middle of the next decade.

---

\(^1\)A value obtained by combining the results from the LEP experiments presented at the Joint International Lepton-Photon Symposium & Europhysics Conference on High Energy Physics, Geneva 1991

\(^2\)Today we know that the mass of the \(Z^0\) is 91.182 ± 0.021 GeV and the W mass is 80.230 GeV ± 0.350 GeV. The rest mass of the proton is 0.938 GeV.
3 The Large Hadron Collider.

The new CERN facility to explore the mass region up to 1 TeV is called the "Large Hadron Collider", LHC. As shown in Fig. 1, there is sufficient space to construct a new superconducting magnet ring on top of the existing LEP magnets. Two beam pipes will be located in one dipole-magnet unit. In these beam pipes bunches of protons will circulate in opposite directions, and they will collide at four points on the ring. Some of the design parameters of LHC are listed in Table 2. A complete listing is found in [5]. The most important of these parameters are the energy, the luminosity and the bunch spacing. The maximum energy of the protons is 7.7 TeV. This value is set by the maximum magnetic dipole field, which can be made available with large scale produced superconducting magnets. With sufficiently high luminosity this is enough to make searches for new particles with masses up to 1 TeV. Luminosity is another word for particle intensity. What sufficient luminosity means, is illustrated in the following.

Let us use the Higgs-particle as our target for the research at LHC. Figure 2[6] shows the production cross-section for the Higgs-particle multiplied by the probability for the Higgs-particle to decay into two Z^0 particles each decaying into pairs of charged leptons. The cross-section is given as a function of the Higgs-particle mass. Four high energetic charged leptons, pairwise having an invariant mass near the mass of the Z^0, will provide a very clean signal for the Higgs-particle. In particular when the Higgs mass becomes significantly larger than two times the mass of the Z^0. The average number of events observed per unit of time from a given process is found as the product of the cross-section, \( \sigma \), the average luminosity, \( < \mathcal{L} > \), and the detection efficiency, \( \epsilon \):

\[
\mathcal{N} = \sigma \cdot < \mathcal{L} > \cdot \epsilon
\]  

(1)

Let us for simplicity assume, full detection efficiency, including all stages of the experiment from
LHC design parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. energy</td>
<td>7.7 TeV</td>
</tr>
<tr>
<td>Luminosity</td>
<td>$1.65 \times 10^{34}$ cm$^{-2}$s$^{-1}$</td>
</tr>
<tr>
<td>Number of bunches</td>
<td>4725</td>
</tr>
<tr>
<td>Bunch spacing</td>
<td>4.5 m, 15 ns</td>
</tr>
<tr>
<td>Number of interaction points</td>
<td>3</td>
</tr>
<tr>
<td>Bunch length</td>
<td>7.5 cm</td>
</tr>
<tr>
<td>r.m.s. beam radius at crossing points</td>
<td>15 µm</td>
</tr>
<tr>
<td>Full crossing angle</td>
<td>200 µrad</td>
</tr>
<tr>
<td>Particles per bunch</td>
<td>$10^{11}$</td>
</tr>
<tr>
<td>Stored beam energy</td>
<td>583 MJ</td>
</tr>
</tbody>
</table>

Table 2: The most important LHC design parameters.

Figure 2: The production cross-section times branching ratio for the Higgs particle in proton-proton collisions as a function of the Higgs mass. The two lines indicate the prediction at 17 TeV (LHC) and 40 TeV (SSC).
trigger to the final reconstruction. We need 1000 Higgs-particles in one year, \(10^7\) sec, to make the discovery of the Higgs-particle with a mass around 250 GeV. With a Higgs-particle mass equal to 250 GeV we read from Fig. 2, that the cross-section is of the order of 0.01 pBarn or \(10^{-38}\) cm\(^2\). By inserting these numbers in equation 1 the required average luminosity is found to be:

\[ <\mathcal{L}> = 10^{34}\text{cm}^{-2}\text{s}^{-1} \]

At the CERN proton-antiproton collider the average luminosity was more like \(5 \cdot 10^{30}\text{cm}^{-2}\text{s}^{-1}\), three orders of magnitude smaller. As a general rule we have, that a process with a cross-section of \(\mathcal{C}\) pBarn will give rise to a trigger rate of \(\mathcal{C} \cdot 10^{-2}\) Hz at a luminosity of \(10^{34}\text{cm}^{-2}\text{s}^{-1}\).

As a consequence of the high luminosity we must handle a high rate of not very interesting events. Figure 3 shows the total cross-section for proton-proton interactions as a function of the center of mass energy of the colliding proton-proton system, \(\sqrt{s}\). The two curves indicate the limits on the theoretical extrapolation from the existing collider results. The total cross-section at LHC, \(\sqrt{s} = 16\) TeV, is expected to fall between 90 mBarn and 130 mBarn. The uncertainty in the theoretical prediction is only of the order of 20%, which is small compared to other uncertainties in this game. By inserting \(10^{34}\text{cm}^{-2}\text{s}^{-1}\) as the average luminosity in equation 1 and 100 mBarn as a good guess for the total cross-section we find, that \(10^9\) interaction will take place every second in the center of the detector. The beams will be bunched, with a bunch-separation of 15 nsec. Hence 15 interactions will occur each time two bunches cross each other. All the elastic events will escape the detector near the beam pipe, so the number of events actually observed in the detector is reduced by 40% to 9 events per bunch crossing, still a very high number. Longer time intervals between bunch crossings will result in more overlapping events. Shorter intervals will make a smaller number of overlapping events, but will make the selection task for the trigger system intolerably difficult.

The high luminosity is also necessary to discover the very heavy vector bosons \(Z'\) and \(W'\), and
Figure 4: The top-quark production cross-section as a function of the top-quark mass. The curves represent the cross-sections at the Tevatron collider, (1.8 TeV), LHC, (16 TeV), and SSC, (40 TeV).

to find Super Symmetric particles. None of these particles are known to exist, but if they do, they will have production cross-sections similar in magnitude to that of the Higgs-particle. Not all important processes are as rare as the production of the Higgs particle. Figure 4 shows the production cross-section for the long searched top-quark as a function of its mass. If the mass is 160 GeV, the production cross-section is $10^3$ pb. At LHC with full luminosity this is equivalent to a production of 10 top-quarks every second. Such a high rate is not necessary to explore the physics related to the top-quark, so even at lower luminosity it is possible to study important physics questions. This will most likely be the case in the beginning, before the new machine is fully operational.

4 A general purpose detector at LHC.

A general purpose detector must be able to recognize electrons, photons, muons, and hadrons. It must be able to measure energy and/or momentum of the different particles. Also non-interacting particles must be recorded through an unbalance in the transverse momentum $^3$. Table 3 presents in a very crude form the accessible physics processes one would expect to cover with various combinations of experimental equipment. The most simple detector only detects the muons produced in the proton-proton collision. All

$^3$The transverse energy of a particle is defined as $E_T = E \sin \theta$. The transverse momentum vector for a particle is then given by $\vec{p}_T = (E_T \cos \phi, E_T \sin \phi)$, where $\theta, \phi$ are the azimuth and polar angle respectively. The sum of $\vec{p}_T$ over all particles produced in a collision is zero. Particles escaping detection will be observed through a non-zero value of $\vec{p}_T$ when summed over all observed particles.
<table>
<thead>
<tr>
<th>Detector signature</th>
<th>Accessible physics process</th>
</tr>
</thead>
</table>
| $\mu^\pm$          | $H \rightarrow Z^0 Z^0 \rightarrow 4\mu^\pm$  
                     | $Z' \rightarrow \mu^+ \mu^-$                  |
| $\mu^\pm, jets, p_T^{miss}$ | add:           
                     | $H \rightarrow Z^0 Z^0 \rightarrow \mu^+ \mu^- \nu \bar{\nu}$  
                     | $W' \rightarrow \mu^\pm \nu$  
                     | compositeness          
                     | $\bar{q} \bar{q}$  
                     | (direct decays and jet spectroscopy) |
| $e^\pm, \gamma, \mu^\pm, jets, p_T^{miss}$  
(non-)magnetic central part  
reduced tracking | add:           
                     | $4 \times H \rightarrow Z^0 Z^0 \rightarrow 4l^\pm$  
                     | $4 \times H \rightarrow Z^0 Z^0 \rightarrow t^+ t^- \nu \bar{\nu}$  
                     | $H \rightarrow \gamma \gamma$  
                     | 2x rate of $Z'$ and $W'$  
                     | $\bar{q} \bar{q}$ (also cascade decays)  
                     | Heavy quarks, Super symmetric Higgs. |
| $e^\pm, \gamma, \mu^\pm, \tau^\pm, b, jets, p_T^{miss}$  
full momentum and tracking | add:           
                     | more redundancy and cross-checks on the above:  
                     | Heavy flavour tags |

Table 3: A very simplified summary of the detector capability versus the accessible physics signature.
other particles are either absorbed in non-active detector elements or leave the instrument undetected. Such an experiment will provide a very clean but sparse signal for the discovery of the Higgs particle by its decay into four muons. A new neutral heavy vector boson $Z'$ decaying into a $\mu^+\mu^-$ pair could also be discovered with this kind of particle dump experiment.

By adding hadronic calorimetry to the detector, one can measure the direction and energy of jets and an unbalance in the transverse momentum, the sign of non-interacting particles in the event. Neutrinos and the lightest super symmetric particle, LSP, will pass right through the detector without any measurable energy deposition. This addition would bring many new possibilities. More channels to the Higgs search. A new charged heavy vector boson $W'$ can be seen. Super symmetric particles can be recognized, and if a substructure of the fermions exists at a 10 TeV scale, compositeness, this can also be studied.

By adding electromagnetic calorimetry and a reduced tracking many extra channels for the Higgs, $Z'$ and $W'$ searches will be available. New heavy quarks and super symmetric Higgs particles can be observed.

The most complete general purpose detector will have full tracking in a central magnetic field in addition to the detector described above. This brings a lot of redundancy and cross-checks to the measurement and the possibility to tag $b$- and $t$-quarks.

Figure 5 shows a possible layout of a general purpose LHC detector. It is the result of design studies in the EAGLE proto-collaboration. This is not a final design but a good and qualified guess to how a general purpose detector could look. Particles produced in the center of the experiment will on their way out traverse silicon strip detectors for vertex reconstruction and some kind of track detectors. The charged particles will be bent in a relatively weak magnetic field created by a superconducting solenoid with its symmetry axes overlapping the beam line. Electrons and photons will deposit most of their energy in the electromagnetic calorimeter. In order to make a precise distinction between electrons, other charged particles and photons, it
is proposed to place a preshower detector in front of the electromagnetic calorimeter. It could consist of a tungsten converter surrounded by silicon strip detectors. Charged particles would give a signal in a few strips in the detector in front of the tungsten, whereas photons would pass through without leaving any signal in the first silicon layers. In the detector after the tungsten converter, charged particles except for the electron would give a small local signal. Electrons and photons will start a shower in the converter and leave a huge signal distributed over many strips. Combining the information will separate the three particle types. Other particles, except for the LSP, the muon and the neutrino, will deposit all their energy in the calorimeters. The muons will deposit a little energy, minimum ionization, in the calorimeters and hits in the muon-chambers positioned on the outside of the detector. Each subdetector will consist of between $5 \times 10^5$ and $5 \times 10^6$ channels, a number which will be determined from a balance between the necessary sensitivity to the physics signal and the cost.

5 Electronics and trigger.

With only 15 nsec between bunch crossings, future trigger systems must make use of a multi level decision architecture and an extensive pipelining.

5.1 The front end electronics and first level trigger

The purpose of the first level trigger is to make a quick selection of events, so that the event rate falls below a level, where more conventional electronics or micro-processors are able to perform a further reduction. A reduction of the event rate from $10^9 s^{-1}$ to $10^5 s^{-1}$ is assumed in most LHC system designs. Each of the roughly $10^8$ calorimeter cells is equipped with a preamplifier followed by a 66 MHz Fast ADC (FADC), Fig. 6. A digital pipeline stores the output from the FADC every 15 nsec. The length of this pipeline is dictated by the decision time of the slowest subtrigger. The subtriggers are

- electron triggers, based on localized energy deposition in the electromagnetic calorimeter. More than one trigger threshold are applied in order to combine this type of triggers with other subtriggers.

- jet triggers, based on localized energy in both the electromagnetic and the hadronic calorimeters. Multi-thresholds are applied.
• missing transverse momentum,

\[ \vec{p}_t = \sum_i \left( \frac{E_i \sin \theta_i \cos \phi_i}{E_i \sin \theta_i \sin \phi_i} \right) \]

The trigger selects on the length of this vector, where the sum is over all calorimeter cells.

• total transverse energy trigger. This is not useful as a stand-alone trigger, because of the contribution from the many overlapping events. But it may be useful in coincidence with other subtriggers.

Model descriptions of a first level trigger show that all subtriggers will use of the order of 1 μsec to make a decision. Hence a 1 μsec or longer pipeline is necessary to satisfy the requirement and at same time to save a few digitizations from bunch crossings prior to the one of interest. In addition to the four calorimeter triggers a muon trigger is vital for the experiment. It must be based on crude vertex information from detectors near the beam pipe and muon chambers behind the calorimeters.

While the first level trigger decision is taken, the fully digitized calorimeter signal moves through the digital pipeline. The content of the pipeline is constantly overwritten until an event satisfies a trigger condition. At that moment the event is shifted in to another memory, where it can be accessed by the processors in the two following trigger stages.

A signal to the readout processor in the second level trigger is queued when an event is accepted by the first level. Zero deadtime is the ultimate goal for the first level trigger.

An excellent example of an effort to put all these features into one hybrid, is the FERMI project [7]. Figure 7 shows the FERMI layout. 9 channels are placed on one hybrid, 8 active and one spare. Each line has a non-linear amplifier, a 10-bit 15 nsec conversion time ADC and a 16 bit lookup table containing calibration constants and correction factors for the nonlinearity of the amplifier. This gives an ADC with an effective dynamic range of 15 bit. The output from the lookup table is routed to the 1. level trigger and through a filter to a pipeline. After an accept from the first level trigger, the event is shifted into one of the second level buffers. Many such FERMI modules can be grouped on one board. The FEAST project is currently looking into this possibility. From the memory located on the FEAST-board, the level 2 and level 3 processors can have access to the data.

The use of artificial neural networks is under study for their usefulness in trigger systems. Two good examples are the D0 electron trigger [8] and a track trigger[9] from Denby et al. Hardware implementations of networks with one hidden layer are under design study in Lund, Sweden. The response time could be as short as 15 nsec, fast enough to be used in the first level trigger. Slower but more flexible chips are available from commercial manufactures, with a response time around 1 μ sec. Slow chips are only useful at later stages of the selection chain, where they can be used as co-processors to RISC machines. In the case of big pattern recognitions problems, i.e. track finding, neural network has already proven to be a very powerful tool.

5.2 The second level trigger, the third level trigger and event building.

The purpose of the second level trigger is to reduce the event rate from \(10^8 \text{s}^{-1}\) to well below 1000 \(\text{s}^{-1}\). In addition to the tools available at level one, are better calibration constants and the possibility to use variable size energy clusters. Crude track information might also be available.

The final stage of the triggering and data acquisition system needs all information in a final and precise form in order to reduce the event rate from 1000 events per second to \(\sim 10\) events per second. In most of the proposed designs for future DAQ systems, the necessary computer power is obtained by connecting a huge amount of RISC-processors in a network, commonly named a farm. A combination of the INTEL i860 RISC machine and one of the new bus architectures SCI[10] or FUTUREBUS+[11] is a good candidate for such a farm. Each processor is given a full event on which the final decision is made and the event built.
FERMI DATA ACQUISITION REQUIREMENTS (DA)

- Multiple (probably 9) parallel data channels
- 15 ns/sample
- 15 bit dynamic range, compressed to 10 bits by a non-linear amplifier, then expanded to 15 bits by a look-up table after digitization
- Digital filtering of signals
- Sum output for first level trigger
- Event recognition
- On module level 1 and level 2 data storage
- No data loss for fixed level 1 processing time
- Low power consumption
- Fault tolerance

Figure 7: The FERMI chip layout. The module contains 9 channels. Each has an amplifier, a 15 nsec ADC, a look up table of gain correction, a pipeline and storage for the second level trigger. Calibration control is a build in unit on the chip.
Another possible solution to the third level and event builder problem is to use a network of transputers. Such a network will be used at the ZEUS experiment at HERA[12].

A transputer is a microcomputer with on chip memory and four serial point-to-point links for communication to other transputers. The instruction set is very restricted and efficient. It also has access to external memory through a 32-bit parallel bus. In the architecture proposed here the memory following the initial pipeline is mapped onto the memory bus of the transputer. An event accepted by the 2nd level trigger is easily copied into the transputer-memory, followed by a memory release. Each transputer has the responsibility to serve a few hundred calorimeter cells. The transputer calculates the total energy, total transverse energy, the total momentum vector and constructs the momentum vectors for energy clusters inside its local region of the calorimeter. This is done with access to the final calibration constants.

Parallel to this, other transputers find tracks in the same region of the detector and yet other find geographically related quantities from other detector components. All transputers are connected in a network through routing chips exemplified in the recently announced INMOS transputer link routing unit, C104. The unit serves 32 fully connected bidirectional links. The unit is made to connect arrays of INMOS/T9000 25 MFLOPs transputers[13]. The T9000 is able to transfer 100 Mbit per second simultaneously in and out, over each of the four point to point links. Messages can be transmitted between transputers located on different routing units. Figure 8 shows a possible arrangement of transputers from geographically related detector components. The local information, relevant for the trigger, is transmitted to the local trigger master, where final clusters are constructed out of subclusters, electromagnetic energy is combined with energy in the hadronic calorimeter. Tracks are related to energy and signals from other components, such as transition radiation and preshower detectors. All is merged into a local trigger package and mailed to a global trigger transputer, as shown in Figure 9. As soon as a local transputer has delivered its trigger information, a new event can be read and processed. The event buffer is only released if the event is rejected, or if the read out has finished. Figure 9 also demonstrates the use of the remaining three transputer links. The four functions, triggering, read out, control and monitoring are each allocated one of the links. Through router links a simple data acquisition protocol can be established. When an event is accepted, it can be read through optical links into traditional processors hosted on a traditional data acquisition bus.

The proposed architecture has the advantage of keeping the readout parallel all the way through
Figure 9: Message routing unit network. The proposed architecture separates the four online functions, triggering, event readout, event buffer management (control) and monitoring.

the system. Only events accepted by the third level trigger are moved to one common place. Even though the data rate is high, none of the links between the second and the third level triggers and internally in the transputer network needs to have a particularly high band width. 100 Mbit per second is more than sufficient in a highly parallel system. Programs, calibration constants and data base information can be loaded through the network. The transputers also provide access to the first stages of the trigger electronics through the memory bus.

Transputers are programmable in most high level languages. Process concurrency is easily handled either directly from C-programs or by a UNIX-like operating system supported by a hardware scheduler. The ideal system would run a real time operating system with most of the functionality known from e.g. OS-9.

5.3 Concluding remarks.

Although the challenge to design and construct an experiment for LHC seems overwhelming, recent developments in commercially available electronics give hope for solutions to the problems. A strong participation from industry is mandatory for the success of the project. The physicist have still a long way to go before a convincing estimate of the trigger rates and reduction factors are established. Two major problems have not been addressed here. The heat dissipation from the millions of chips scattered over the surface of the detector and the radiation hardness of the electronics. Both are major issues in future R&D projects.

The battle will be hard, but fruitful to any one who wants to participate in the game.
References

The Electroweak model.
Proceedings of CERN school of physics 1984
CERN 85-11 page 277-299

Observation of neutrino like interactions without muon or electron in the Gargamelle neutrino experiment
Physics Letter 46B(138)1973


Proceedings of the Workshop on Physics at Future Colliders.
LaThuile 7-13 January 1987
CERN 87-07 Vol. 1.

[5] The LHC Study Group
Design Study of the Large Hadron Collider (LHC)

Standard Model Physics at LHC
Proceedings Vol 1. of The Large Hadron Collider Workshop.
Aachen, 4-9 October 1990.
CERN 90-10

[7] A Digital Front-end and Readout Microsystem for Calorimetry at LHC.
R&D proposal.
CERN/DRDC/90-74, DRDC/P 19.

[8] D. Cutts et. al.
Application of Neural Networks in High Energy Physics.
Proceedings of Computing for High Luminosity and High Intensity Facilities.
Santa Fe, NM 1990. Page 203.

Tests of Track Segment and Vertex Finding with Neural Networks.
Proceedings of Computing for High Luminosity and High Intensity Facilities.
Santa Fe, NM 1990. Page 211.


Proceedings on "Workshop on New Backplane Bus Architecture"
Cern 22-23 March 1990. CERN-CN/90/4
    Ronald Boontje, Transputers in Data Acquisition at NIKHEF used in Hadron Experiments.
    L. Wiggers, Data Acquisition for the ZEUS Calorimeter Test Set Up at CERN
    Henk Uijterwaal, The ZEUS Global Second Level Trigger.

    INMOS document DBTRANSPST/1
    The Transputer Databook
    The Transputer Development and IP Systems Databook
Collisions at future Supercolliders: the first 10 microseconds.

L. Mapelli

CERN, Geneva, Switzerland

Abstract

The next step forward of the fundamental research in Particle Physics is programmed via the construction of Supercolliders. By this term, the High Energy Physics community indicate the next generation of accelerators consisting of very high energy and very high intensity proton-proton colliders. The physics expectations from these machines include the completion of the picture depicted by the Standard Model and/or the unveiling of new phenomena capable of changing the scenario of today's theoretical understanding of the basic laws of Nature. Such a grandiose scenario implies an unprecedented level of technical difficulties for the experimentation at these new generation of machines. Many of the technical problems are still unsolved today and an intensive R&D program is under way. Some of the problems are related to techniques and architectures for the Trigger and Data Acquisition systems, i.e. to the capability to effectively select and record data from interesting interactions. These lectures are aimed exactly at illustrating the challenge of Triggering and Data Acquisition at Supercollider experiments, presenting the status of the art of the present technology and indicating, where appropriate, the nature of the necessary Research and Developments.

1. INTRODUCTION

The next significant step towards a better understanding of the laws of Nature is expected from the study of particle interactions at energies in the TeV region. Today's accelerator technology is believed to be mature for the realisation of multi-TeV hadron colliders, capable of providing TeV interactions at the constituent level. Following the rich results of the successful experimentation at the present proton-antiproton colliders (the CERN Collider and the Tevatron at Fermilab), the High Energy Physics communities of both sides of the Atlantic are concentrating their efforts on the study and the design of the next generation of hadron colliders – the Large Hadron Collider (LHC) at CERN and the Superconducting Super Collider (SSC) in Texas.

Such machines, capable of responding to the physics case presented in sect. 2, will produce very high energy proton-proton collisions at luminosities ranging from $10^{33}$ to $4 \times 10^{34}$ cm$^{-2}$ s$^{-1}$. For the predicted inelastic cross sections – 60 mb at LHC (16 TeV) and 90 mb at SSC (40 TeV) –, interactions will occur at rates from $6 \times 10^7$ to $4 \times 10^9$ Hz. Given the bunch structure of the machines, with bunch spacing of 15 ns (16 ns for the SSC), the collision rate will be 67 MHz (62.5 for the SSC), with an average of 1 to 40 overlapping events every bunch crossing.
In order to cope with the high particle rate and maintain the required identification power, detectors will have high granularities and a general purpose experimental apparatus could have as many as $10^7$ to $10^8$ electronics channels. Present estimates indicate prompt trigger rates of the order of $10^4$ to $10^5$ Hz, implying data acquisition bandwidths of the order of Gbytes/sec. Such machine and detector parameters, an evaluation of which is presented in sect. 3, impose severe demands on the performance of the front end electronics, trigger and data acquisition systems.

A multi-level trigger and data acquisition system is necessary to stand the unprecedented event rates, trigger rejections and data volumes. Following the prejudice coming from our present experience and waiting for the possibility of a more detailed study which needs a full detector design, a three level model, as described in sect. 4, can be used to address the major issues of triggering and data acquisition for the hadron supercolliders. This course addresses the aspects of the first levels, where fast triggering and event selection must assure rejections of several orders of magnitude in an average time of 10 to 100 microseconds (which suggested the title of the lectures). A discussion of the fast trigger levels is presented in sect. 5.

The accelerator parameters together with cross section predictions are used to evaluate the system requirements, based on a general purpose detector model. The available technology is briefly reviewed with the aim of discussing how a trigger and data acquisition system might be constructed, presenting the state of the art of the relevant techniques and indicating where areas of rapid technological advance might influence the system design.

Today's technology is inadequate for the realisation of a system capable of meeting the LHC/SSC requirements, which are reviewed in sect. 6 and compared to the present technological achievements. Traditionally, the preparation of new generation experiments has concentrated on intense developments activities for detector techniques. For LHC/SSC, the need for a major research and development effort also in the field of electronics, triggering and data acquisition is being more and more recognised by the High Energy Physics community and several Research and Development projects have been recently initiated in the context of LHC. A brief overview of the relevant R&D activities, in sect. 7, concludes this course.

Finally, the reader unfamiliar with High Energy Physics jargon is invited to have a look at the Appendix, where he can find a brief definition of the most common technical terms used in this paper.

2. THE PHYSICS CASE

2.1 Theoretical aspect

In the early 80's, the W and Z particles were observed at CERN by the UA1 [1] and UA2 experiments [2] running at the CERN Collider. The discovery of the long sought Intermediate Vector Bosons (IVBs), carriers of the electroweak force, was a beautiful confirmation of the Standard Model of the electroweak theory [3] marking a significant step towards an unified description of the fundamental forces governing the Universe.

Despite the numerous great successes of its predictions, the Standard Model is not a full theory, because it does not explain many fundamental questions of
the Elementary Particle world, of which examples are the origin of the particle masses and the hierarchy of generations of quarks and leptons. The mechanism responsible for the symmetry breaking for the mass generation. A possible solution to at least some of the problems comes from the Higgs mechanism [4], which has been suggested as the one responsible for the electroweak symmetry breaking and the consequent mass generation of the elementary particles.

In the minimal form, the Higgs mechanism requires the existence of one scalar boson: the Higgs. The obvious next step towards unveiling the solution chosen by Nature is, therefore, the experimental hunt for the Higgs boson. Compared to the case of the IVBs, the Higgs search presents much greater experimental difficulties:

- *the mass is not calculable:* it could be anything between the present experimental lower limit of 57 GeV [5] and a theoretical upper limit of almost 1 TeV,
- *the couplings are calculable:* just to show that, being very small, the rate of production is very low
- and ... its existence is not guaranteed at all!

Such a situation leads to the consequences that:

- the energy range for the search has to be as wide as possible,
- the rate of interactions (luminosity) must be as high as possible
- the experimental apparatuses must be 'general purpose' to be sensitive to a variety of signatures at various particle energies and accurate enough that an eventual negative result of the search can guarantee the non-existence.

The search for the Higgs is the main, but not the only, physics motivation for the next generation of experiments in high energy physics: Higgs or no Higgs, our theorists are unanimous in concluding that, at an energy scale of about 1 TeV, 'something' must happen: either the Higgs boson is found or the perturbative theory breaks down and one should see, via the measurement of gauge boson pairs cross section, the onset of a new non-perturbative regime, where the weak interactions become strong [6].

On top of this physics motivation, already strong enough to justify the required investment, the LHC potentiality includes:

- the search for new forms of matter, such as Super-Symmetric partners to the known particles, as well as lepto-quarks or excited leptons
- the study of the composite nature of the elementary particles to an unprecedented level
- and the possibility of a deeper understanding of QCD.

Also one should not forget the unforeseeable surprises that an unexplored energy domain might reserve for us.

### 2.2 Experimental aspect

The 1 Tev energy domain could in principle be reached with electron-positron head-on collisions. A linear collider of this kind (CLIC [7]) is presently under study and would have the great advantage of generating a far easier experimental situation, but the technology necessary for such acceleration is still a
few years ahead of us. On the contrary, accelerator physicists know how to accelerate protons to energies capable of producing components interactions in the TeV energy region: almost 16 TeV for the Large Hadron Collider (LHC) [8] and 40 TeV for the Superconducting Super Collider (SSC) [9]. Unfortunately, in order to reach the sensitivity required by the physics the proton-proton interactions must occur at an unprecedented rate, 2 to 3 orders of magnitude higher than the ones reached so far. Again, today's accelerator technology is mature for that, but at the price of a very short time between successive interactions (15 ns) and of many independent interactions superimposing to each other in a single 'event' (up to 40 for the highest luminosity foreseen).

One does not need to be an expert in particle physics experimental techniques to realise that such accelerator parameters constitute severe difficulties for the experimentation at the LHC and the SSC machines. It is enough to consider that the 15 ns inter-bunch interval corresponds to a distance of 4.5 m covered by a particle who travels at the speed of light, and to a length of just over 2 m of cable covered by electronics signals, while the typical size of a detector for LHC/SSC is several meters in all directions [10,11]. In other words, many successive interactions will occur while the particles from a given event are still flying in the apparatus, while signals are still forming in the detectors and while electronics pulses are still travelling in the cables.

In fact, today's technology is not mature to completely develop a full experiment for the LHC and the SSC, both for detectors and for readout, trigger and data acquisition electronics.

3. PARAMETER OVERVIEW

In order to establish the framework that signal processing, triggering and data acquisition will have to face at the LHC and the SSC, an analysis of the expected trigger and event rates and of data volumes must be performed. In order to stress the extent of the problem, we will consider the case of a general-purpose detector sitting on the machine with the highest luminosity, the LHC. It is understood that whenever, for the rest of the paper, only the LHC is mentioned is just for sake of simplicity, all consideration and conclusion being valid for the SSC as well.

A summary of the machine and experiment parameters which best illustrate the problem of experimentation at LHC is shown in Table 1, where the comparison to the existing experimental situation is also shown. The evaluation of some of the experimental parameters of the table is contained in the following sections.

At this stage any evaluation is necessarily very rough leading to enormous uncertainties in the conclusions. This parameter overview should be considered only as an exercise to establish a working hypothesis.
Table 1. The LHC/SSC jump

<table>
<thead>
<tr>
<th></th>
<th>pp Colliders</th>
<th>LHC/SSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bunch Xing</td>
<td>4 μs</td>
<td>15 ns</td>
</tr>
<tr>
<td>Luminosity</td>
<td>$10^{31}$ cm$^{-2}$ s$^{-1}$</td>
<td>$10^{34}$ cm$^{-2}$ s$^{-1}$</td>
</tr>
<tr>
<td>Event rate</td>
<td>$10^4$ Hz</td>
<td>$10^9$ Hz</td>
</tr>
<tr>
<td>Prompt trigger</td>
<td>100 Hz</td>
<td>$10^5$ Hz</td>
</tr>
<tr>
<td>Physics reduction</td>
<td>$10^{-9}$</td>
<td>$10^{-13}$</td>
</tr>
<tr>
<td>No. channels</td>
<td>$10^5$</td>
<td>$10^5$</td>
</tr>
<tr>
<td>Raw data volume</td>
<td>$10^7$ B/s</td>
<td>$10^{11}$ B/s</td>
</tr>
<tr>
<td>Final data rate</td>
<td>100 kB/s</td>
<td>100 MB/s</td>
</tr>
<tr>
<td>No. physicists</td>
<td>100</td>
<td>$10^3$</td>
</tr>
</tbody>
</table>

3.1 Event rate

Event rates are calculated from the estimated total inelastic cross section at centre-of-mass energy $\sqrt{s} = 16$ TeV. As shown in fig. 1 extrapolations of existing pp collider data [12] suggest the value $\sigma$(tot) = $110 \pm 20$ mb for the total cross section at $\sqrt{s} = 16$ TeV, which corresponds to a visible cross section in the experiment of $\sigma$(inel) ~ 60 mb. For luminosities in the range $10^{33}$ to $4 \times 10^{34}$ cm$^{-2}$ s$^{-1}$, the rate of visible interactions will be in the range $6.7 \times 10^7$ – $4 \times 10^9$ interactions/s. However, given the bunch structure of the beam with bunch crossings every 15 ns, the relevant rate will be $6.7 \times 10^7$ bunch crossings/s with between 1 and 40 overlapping events per bunch crossing.

3.2 Event size

The average event size is extracted from a model of a general-purpose detector, i.e. a detector capable of electron, muon and jet identification and of missing transverse-energy measurement. Such a detector would consist of inner tracking devices with electron identification power, electromagnetic and hadronic calorimetry and muon chambers. The pseudorapidity coverage required for most of the physics signals is $|\eta| < 3$. The measurement of the missing transverse energy requires a bigger coverage, possibly to $|\eta| < 5$. The granularity of these forward detectors will be coarse, thus providing a negligible contribution to the overall event size. A general-purpose detector of this kind will be located in a "medium" luminosity area, where peak luminosities up to about $2 \times 10^{34}$ cm$^{-2}$ s$^{-1}$
are expected. The following estimates are therefore based on a luminosity of $10^{34}$ cm$^{-2}$ s$^{-1}$.

![Graph showing the relationship between energy and cross-section](image)

**Fig. 1.** From the extrapolated cross-section to the expected event rate.

From a parametrisation of the energy dependence of the charged particle multiplicity, $n(ch)$, using CDF results at 0.63 and 1.8 TeV [13], the expected number of charged particles per unit of rapidity at $\sqrt{s} = 16$ TeV is $dn(ch)/d\eta = 6$ for $\eta = 1$. Consequently, the expected total number of particles in the calorimeter is 12 per unit of rapidity. Studies of the rate of particles reaching the muon chambers [14] indicate that for an absorber thickness of 12 absorption lengths, only one track will be seen in the muon chambers for $\sim 10^4$ incident charged hadrons. Hence, at a luminosity of $10^{34}$ cm$^{-2}$ s$^{-1}$ and for a detector coverage of $|\eta| < 3$ one expects:

- $\sim 350$ tracks/15 ns in the inner detectors,
- $\sim 700$ particles/15 ns in the calorimeters,
- $< 1$ tracks/15 ns in the muon detectors.

### 3.3 Trigger rate

Monte Carlo physics results are used to estimate trigger rates. Prompt triggers will be based on signatures of electrons, muons, jets and missing transverse
energy, based on the rough particle identification achievable in the shortest possible time with information from calorimetry and fast muon detectors.

The basic selection parameter at the first trigger level will be a threshold on the transverse energy or momentum. Summarising the results presented in ref. [15] it appears that the expected inclusive rates [16] for a luminosity of $10^{34} \text{cm}^{-2} \text{s}^{-1}$ and a coverage of $|\eta| < 3$ will be dominated by triggers requiring electrons. Assuming that thresholds can be kept reasonably sharp, it seems possible to provide prompt triggers with sufficient acceptance for physics at rates of the order of $10 \text{kHz}$. Rates might be reduced further by demanding more complicated signatures. It should be noted that an inclusive missing transverse energy trigger, if implemented, might also give a high rate. We assume a total first-level trigger rate of $10^4 - 10^5 \text{ Hz}$.

After the fast ($\sim 2 \mu\text{s}$, see sect. 4 for an evaluation) first-level trigger, higher levels of trigger are needed with increasing background rejection power. Tracking information combined with more refined calorimeter cuts for electron identification, precise transverse energy cuts on muon candidates, and kinematic and topological selections can be applied by second-level trigger processors. A rejection compared to the first-level trigger of at least $10^2$ should be possible, leaving an overall rate after the second-level trigger of $10^2 - 10^3 \text{ Hz}$.

Further reduction has to be gained to meet data storage and off-line analysis capabilities. No detailed evaluation of third-level trigger algorithms has been done so far. However, it seems unavoidable that full event reconstruction and on-line filtering probably performed by general-purpose processor farms must reduce the data rate to acceptable levels.

### 3.4 Data volumes and Bandwidths

We can now combine these parameters to predict the overall data volumes and estimate the required bandwidths. An accurate estimate of the data volumes that a data acquisition system will have to face could only be done if the detector configuration were known. Without that, only an educated guess can be tried based on the number of electronic channels of typical subsystem examples after taking into account the expected occupancy. The granularity of the detectors under study for the future colliders varies between a few times $10^5$ channels for the calorimeter options and up to a few tens of millions channels for some inner tracking detectors. Combining the numbers of channels, with their expected occupancies, and the trigger rates evaluated above, one obtains the following estimated required bandwidths:

- **First level:** $10^{10} - 10^{11} \text{ bytes/s}$
- **Second level:** $10^8 - 10^9 \text{ bytes/s}$
- **Third level:** $10^7 - 10^8 \text{ bytes/s}$

As described in more detailed later, the enormous amount of data corresponding to the first-level trigger accepts might not need to be moved out of local memories. A possible scheme, in which the first-level trigger is used to drive the second-level algorithms, would require that only sections of the detector which are flagged by the first-level trigger are read by the second-level processors, leaving the bulk of the data in local storage until the second-level trigger decision is made.
The rate of second-level triggers is such that bandwidths of the order of Gbytes/s are needed in order to empty the buffer memories, build the events and transfer the data to processor farms for higher-level filtering and data storage. Buses, networks and data links capable of such throughputs are discussed in ref. [17].

The 10–100 MB/s rate for data storage has been estimated by arbitrarily assuming a further reduction of a factor of 10 at the highest-trigger level. No systematic study has so far been done on how to achieve such a rejection. However, it is common prejudice that enough processing power will be available in a processor farm to perform quite advanced physics analysis in order to reach a manageable level of data storage.

4. A TRIGGER/DAQ ARCHITECTURE MODEL

Most of the general issues of signal processing, triggering and data acquisition for the LHC have been addressed at successive workshops [18,19,20,21]. Irrespective of the detailed form that a trigger and data acquisition system might have in a LHC/SSC experiment, the boundary conditions described in sect. 3 impose a few basic features to be satisfied to cope with the expected requirements:

- the system must be designed for the highest performance
- the first-level trigger processors, whether analogue or digital, must be pipelined to handle a new event every 15 ns
- pipelined buffering and hierarchical data collection are essential
- where possible, the system should be characterized by a high degree of parallelism for easy scaling and adaptation to evolution of required performance.

In the following, a somewhat detailed explanation of how these requirements come about is presented.

Following experience from experiments at the SPS [22,23] and Tevatron [24,25] proton-antiproton colliders, and from the preparation of the future e–p collider HERA [26,27], one can envisage a trigger system for experiments at the LHC [28] and the SSC [29] based on several levels. Relatively crude decisions made quickly at the first level can be refined at the second and third levels using more detailed information from the detectors and more complicated algorithms. The role of each trigger level is to reduce the trigger rate to the point where it can be accepted by the next higher level. Fig. 2 illustrates the structure of such a trigger system, based on three levels, with an indication of order-of-magnitude rates which might be achieved after each level of triggering.

It must be stressed that this architecture only reflects current ideas for a powerful trigger and data-acquisition system. It is not appropriate to try to define a final solution for trigger and data acquisition now, since we do not yet have at our disposal the technology needed, especially in the front-end.

The size of a typical LHC/SSC detector is ~30 m long by 20 m across. For such a large detector, the time to form a first-level trigger decision is of the order of 0.5 μs, allowing only for cable delays to bring information from the whole detector to one central place and then to send the trigger decision back to the electronics mounted on the detector, i.e. the trigger processing time has to be added.
Given that the first-level decision time must be at least several hundred nanoseconds and since a bunch crossing occurs every 15 ns, information for many events must be stored for each channel during first-level processing; each event may contain several interactions. Storage devices known as pipelines are expected to fulfil this role. The pipeline memories, which may be analogue or digital-storage devices, accept new information every 15 ns. The data are stored until the first-level decision is available, after which they are either discarded (first-level reject) or transferred to the second-level (first-level accept). The number of storage elements in the pipeline for each detector channel is determined by the first-level decision time. For each microsecond of first-level decision time, 67 storage elements are required.

The first-level trigger will take more than 1 μs and may take up to ~2 μs to deliver its verdict to the detector electronics, this latency being made up of a combination of the response times of detectors and their associated electronics, cable delays and propagation delays through trigger electronics. However, the trigger must be able to start analyzing a new event every 15 ns. We must therefore think of a trigger system concurrently processing many events each separated in time from the next one by 15 ns. The idea of pipelined processing, commonly employed in high-performance computers, can be adopted.

When an event is accepted by the first-level trigger, the data from the whole detector are moved into buffer memories and stored during the second-level trigger processing. The model presented here foresees digital second-level buffers, so for detectors with analogue first-level pipelines an ADC must be included in
the scheme as shown in fig. 3a. The alternative of prompt digitization and a
digital first-level pipeline is shown in fig. 3b.

Fig. 3 (a,b). Second-level buffer memory with analogue (a) or digital (b) first-level pipeline.

These two alternatives can coexist, some subdetectors using analogue and
others using digital pipelines. However, independently of whether or not the
storage elements are analog or digital, all first-level pipelines from all
subdetectors must be synchronized. This essential consideration, which is already
important in HERA detectors, means that a detector-wide view must be taken
before detailed design of the readout electronics for individual subdetectors.

The second-level trigger system must access information from the detector.
There is therefore an intimate relationship between second-level trigger
processing and second-level data storage. One extreme view is to consider the
second-level buffer as an integral part of the second-level trigger and to move all
the data from the detector (or subdetector) into the trigger processor system. At
the other extreme, one could buffer the data on the detector, only moving those
data which are needed by the second-level trigger processors.

For some subdetectors very large volumes of data are involved – perhaps
10^5 bytes per event for a calorimeter and even more for a central tracking detector,
even after some data compression (or zero suppression) has been performed.
Moving such large amounts of data after each first-level trigger will not be easy
given trigger rates up to 10^5 Hz. Where possible, it may be preferable to buffer the
data locally until after the second-level trigger has accepted an event. Note that
for triggers in which the processing is localized to a small part of the detector,
such as electron or muon triggers, the first-level trigger can be used to flag regions
of the detector containing candidates. This information can then be used by the
second-level trigger, avoiding the need to access data from the whole detector.

Thinking of second-level trigger systems loosely associated with specific
detectors, it may be important to combine information from different detectors in
the second-level trigger. Detailed rate calculations for second-level triggers are
still at an early stage. However, it is believed that the second-level trigger should
be able to gain a factor of more than 10^2 in rejection by refining the first-level
trigger decision using more precise information from the detectors, and by
combining information from different detectors. Thus, the rate from the second-
level trigger will be in the region 10^2-10^3 Hz. Note that the information available
at this stage need not be crude: if up-to-date calibration is available, electrons, muons and jets could be measured using the full precision of the detector.

In the model shown in fig. 2, events selected by the second-level trigger are fully read out before third-level processing is performed. The second-level trigger will already have made sophisticated decisions based on information from several (maybe all) subdetectors. Third-level processing may have to do full event reconstruction and make physics analysis cuts: at a luminosity of $4 \times 10^{34}$ cm$^{-2}$ s$^{-1}$, the predicted rate for $W \rightarrow e\nu$ and $W \rightarrow \mu\nu$ decays is hundreds per second!

5. TRIGGER

From the general physics requirements of sect. 3, the trigger system must select at level 1 a set of events for the physics analysis and provide progressively more powerful selections at higher levels. As a general introduction to the trigger discussion, fig. 4 shows the expected time structure of the three-level trigger model used here.

![Time structure of the LHC three-level trigger model.](image)

Similarly, fig. 5 shows a schematic summary of the equivalent time structure of the multi-level trigger systems of typical today's experiments, which has inspired the LHC model.

The comparison between the two figures gives already an indication of the much greater difficulty of experimentation at LHC and SSC because of the many order of magnitude higher event rate and shorter machine crossing time.

Using the three-level model of the trigger introduced in sect. 4, here we discuss how first and second-level triggers based on calorimeter or muon-detector information could be implemented. Triggers based on other detectors, probably used to refine the electron signature at the second-trigger level, are also discussed. The content of this section is largely based on the equivalent one of ref. [28].
5.1 Calorimeter triggers

As discussed in sect. 3, calorimeter triggers will have to provide the basic prompt selection of high \( p_T \) electrons and photons, jets, and also events with missing transverse energy. The isolation of electrons and photons can provide an additional handle for separating interesting physics from the dominant background of jets.

5.1.1 First-level calorimeter trigger

Prompt (level 1) calorimeter triggers can be implemented using either analogue [30] or digital electronics [31]. A trigger for high-\( p_T \) electrons (or photons) could be implemented by a system of discriminators attached to the front-end electronics of the electromagnetic calorimeter. One would have to make an analogue sum over samplings in depth (if more than one), and might in addition perform lateral summation. Even if one wishes to retain the full granularity of the calorimeter, it is desirable to form overlapping windows so that showers which share their energy between calorimeter-cells trigger with good efficiency. Such a system was implemented for the UA2 experiment [32]. Summing over larger areas of the calorimeter has the advantage of reducing the number of channels to be handled by the first-level trigger. It should be noted that the pipelining of the trigger operations, necessary to handle the 15 ns bunch-crossing period, might constitute a severe problem for an implementation analogue components.
The identification of isolated electrons in the first-level trigger is potentially very useful, giving a substantial rejection against the background from jets. This is not easy to implement using analogue electronics, although resistive networks could provide a solution.

A trigger for high-\(p_T\) jets might also be possible by making analogue sums over jet-sized areas of the calorimeter, and discriminating on the sum. For this, analogue information from the electromagnetic and hadronic calorimeters would have to be combined. In such a scheme it is important to control carefully the relative calibration of the different calorimeters which enter the analogue summing logic, otherwise the resolution will be degraded and the trigger threshold smeared out.

The problem of implementing a missing transverse energy trigger using analogue electronics looks very hard. Whereas electron and jet triggers can be implemented with the electronics mounted locally on, or near to, the detector, missing transverse energy requires a vector sum of transverse energy extended to the entire detector, although the first stages of the sum could be done locally.

An alternative approach to first-level calorimeter triggering is to use a digital trigger processor. This technique, which is illustrated in fig. 6, has already been used in a number of experiments.

One generally makes an analogue sum of a small number of calorimeter channels, before digitizing, using a fast ADC. This sum combines the different depth samplings of the electromagnetic calorimeter and generally involves some lateral summation as well, typically over an area of \(\Delta\eta \times \Delta\phi = 0.1 \times 0.1\). This reduces the number of channels to be digitized for the first-level trigger. It is desirable to make independent digitization of the hadronic calorimeter so that calibration differences between the different calorimeters can be corrected. This also makes possible the implementation, in the first-level trigger, of electron identification based on the longitudinal profile of the shower, as well as lateral isolation.
It is worth noting that a more ambitious approach [33] is to fully digitize all channels from the calorimeter before the first-level pipeline. If this were done, separate digitization for the first-level trigger would be unnecessary. This would allow the trigger to use information from the detector calibrated channel by channel.

The principle of operation of a pipelined digital processor is illustrated in fig. 6(b). This logic, which would be repeated for every electromagnetic trigger channel in the calorimeter, sums over two-by-two (overlapping) windows and compares the sum with a threshold. The latches control the flow of data through the processor; three pipeline steps are included between the input to the first adder and the output of the logic, corresponding to 45 ns. Much more elaborate logic would be used in a practical processor. The clustering of cells in two-by-two windows and the organisation of the windows for isolation and shower development cuts is illustrated in fig. 7.

![Cell arrangement for an electron trigger.](image)

A major advantage of the digital compared to the analogue scheme is that it is possible to make more complicated decisions at the first level. A digital processor can offer electron triggers with overlapping sliding windows, several thresholds with different multiplicity requirements, and (optional) isolation requirements. For jet triggers, the relative calibration of the electromagnetic and hadronic calorimeters can be adjusted. Most important of all, a full missing transverse-energy calculation can be implemented. In addition, a digital implementation gives easy control of the calibration, good monitoring and lots of flexibility.

Preliminary studies for a trigger processor offering all the features described above look encouraging. Simple calculations give a total latency of the level 1
processing (including cable delays for a typical LHC/SSC detector size) of a bit more than 1 μs. Recent technological advances make custom chip design much more accessible to us than a few years ago. Even very complicated processors could be made using only a few custom chip designs, each containing a large amount of logic. A single custom chip could in the future perform functions which in previous digital processors occupied almost a whole circuit board. This gives higher speed and better reliability; it can also be very cost effective. As in existing systems, the number of interconnections will be a serious problem. Thankfully, custom chip packaging now allows very large numbers of external connections.

Of course, there are many considerations which have to be taken into account when comparing analogue and digital solutions for first-level triggering. The amount of power dissipated by digital electronics may be larger than that for an analogue system; for some calorimeters this may be an important consideration which is related to the question of where one should locate the electronics and also to problems of cabling. Also relevant is the issue of radiation hardness of any electronics which is installed inside the detector; it is worth noting that radiation-hard digital and analogue electronics are now available to us from industry. While a digital solution may require larger numbers of connections (several bits in parallel instead of a single analogue signal), there are not the same problems with noise.

Although the implementation of synchronized pipelined processing distributed over the area of the detector is not going to be easy, either for digital or for analogue first-level calorimeter triggers, both solutions look possible in principle, at least for electron triggers (see sect. 6.5 for a brief discussion on the subject). The best choice may depend on the calorimeter technique selected; equally, the choice of calorimeter technique should be influenced by how well the calorimeter can be used in the trigger.

5.1.2 Second-level calorimeter trigger

The speed required for the prompt (level 1) trigger is incompatible with use of programmable processors imposing the implementation of predefined 'hardwired' algorithms with ad-hoc processors where only the value of a few parameters can be varied. On the contrary, the longer time available for the second-level trigger allows one to use fully programmable devices, where one would have total freedom in algorithm implementation and subsequent modification, compatibly with the average time available per event. There is a large variety of commercial processors from which to choose. High-performance general-purpose processors, including Reduced Instruction Set Computers (RISC), are the most flexible and easiest to program. Digital Signal Processors (DSPs) offer more computing power for certain applications [34]. Parallel computers (e.g. transputers [35], distributed array processors or associative string processors [36]) are an even more powerful alternative for problems which can be solved by a parallel algorithm. Image processors, used in the television industry, may also have a role to play; new developments for high-definition television (HDTV) may be particularly relevant here, as discussed in sect. 6.6. One should also mention that neural networks [37] could offer an alternative to traditional computing techniques, and their use might even extend to level 1 triggering.

The second-level trigger system is more than just processing power. Equally important are the data links, buses or networks which allow the processors to
access the data. The overall architecture of the second-level system — buffer memories, data transmission systems and processors — must be considered as a whole. Different problems require different architectures, particularly depending on whether the problem is a local one (such as finding clusters in a calorimeter) or a global one (such as calculating the missing transverse energy).

The interaction between the first and second-level triggers is also relevant. If, for example, the first-level trigger has already identified the location of all candidate high-$p_T$ electrons, this information can be used by the second-level trigger [38]. This is illustrated in fig. 8 which shows a system in which local processors, distributed over the detector, validate electron candidates identified by the first-level electron trigger. In this case, the need for extraordinary processing power and data-transfer rates is avoided, as described below.

![Diagram](image_url)

**Fig. 8.** Example of a second-level architecture for an electron trigger. The ovals represent trigger processors.

Assume that the entire calorimeter is mapped onto, let us say, 1000 local processors, each therefore seeing only 1/1000 of the detector. If a level 1 trigger signals the existence of electron candidates only to the relevant processors, each of them would have to respond in average to only $10^2$ Hz for a total level 1 rate of $10^5$ HZ. There must, in addition, be global processors which gather together information from the local processors before making an overall decision. These could be implemented as a processor farm, with ~ 100 processors taking turns to process events; each global processor must then respond to 1% of the first-level triggers, giving a rate per processor of 1000 Hz. Thus, the time scale for processing an event is ~ 1 ms instead of ~ 10 μs required if different events are not processed in parallel. Given the processing power of today's microprocessors, very sophisticated algorithms can be executed given an ~ 1 ms time scale.

The data rates in such a second-level architecture must also be considered. Suppose each local processor in the above example has to access an area of the detector corresponding to ~ 2000 calorimeter cells (~ 1% of the calorimeter) in order to validate the electron candidate. Then the data rate into each local processor is only ~ 0.4 MB/s. Very little data need to be sent from the local processors to the global processors — just a few words per candidate electron ($E_T$, ...
\[ \eta, \phi, \chi^2 \] — with a first-level trigger rate of \( 10^5 \) Hz and \( \sim 100 \) bytes per event, this corresponds to a total of only 10 MB/s into the farm of global processors, which is not far from the typical bandwidth of standard VME.

Thus, by making use of information from the first-level trigger about the location of candidate electrons and by moving only those data which are required, one can envisage an architecture in which each processor has a long decision time and in which data-transfer rates are modest. The full data from the events must, of course, be stored somewhere during the second-level trigger processing; this can be done locally. The requirement that the second-level processors should selectively access the data, must be included in the design of the buffer memories; a system which includes this possibility is under study [39]. The communication between the buffer memories, the local processors and the global processors is also not trivial despite the low data transfer rates.

The complication of a trigger architecture as the one described and the necessity of a coherent view of all its elements – buffer memory, local and global processors, data links, imposes a detailed simulation for a proper design and the comparison of different implementations. Simulation programs such as MODSIM, VERILOG or ARTIFEX [40] allow the performance of a system to be studied as a function of parameters such as buffer depth or algorithm execution time. It also allows one to study the response under more realistic conditions by simulating error conditions: in some existing experiments error recovery is a dominant source of dead time.

It must be made clear that the architecture described above for an electron trigger is only one possibility. Alternative schemes using massively parallel processors and image processors have also been, and still are, studied [36]. These studies have concentrated on the comparison of different computer architectures for cluster analysis.

The problem of making a second-level missing transverse energy trigger is very different from that of finding or validating electron candidates. One needs to access data from the whole detector, but the algorithm is very simple and well defined: essentially a weighted sum over calorimeter cells. This could already be done by a digital first-level trigger.

It must be remembered that the second-level trigger must be powerful enough to gain a factor of \( 10^2 \) in rate compared to the first-level trigger. Simulation studies for second-level algorithms should take account of the fact that the first-level trigger may already have used many of the easy signatures. Ideally, the first and second-level simulation should be integrated, although the many order of magnitude different time scale might prevent the efficient implementation of a global model.

5.2 Muon triggers

Muons are characterised by their ability to penetrate thick layers of material. Muon identification is, therefore, performed by means of detectors external to the thick hadron calorimeter, which also act as a hadron absorber. The physics requirements for LHC and SSC experiments [15] demand triggers on high-\( p_T \) muons, which are further identified by the small angular deflection in strong magnetic fields. Such triggers can be implemented using information from external muon detectors which are shielded from the interaction region by many interaction lengths of material [41]. These detectors provide several-position
measurements along each track, with good precision in at least one coordinate. The detector must provide fast signals for first-level triggering; this requirement must be considered in the detector design. At the second level it will be necessary to measure accurately the momentum of very high-$p_T$ muons, possibly requiring a detailed analysis of the track trajectory in the magnetic field.

5.2.1 **First-level muon trigger**

A first-level muon trigger can be based on logic which compares the pattern on hits in fast muon chambers with patterns which are valid for high-$p_T$ muons that originate from the interaction region. Several techniques are possible [42] such as look-up tables stored in RAMs or programmable logic; both of these are very fast ($\sim 15$ ns). The result is a flag indicating the presence of a muon candidate: the first-level trigger provides the position of candidates but no information on their transverse momenta.

Thresholds can be controlled by changing the list of valid hit patterns. Essentially, one defines roads from the interaction region through the muon chambers in which one requires hits. Narrow roads correspond to high-$p_T$ muons, wider roads to lower-$p_T$ muons. While the implementation of such a trigger system appears to be relatively straightforward, the details depend on the geometry chosen for the muon detector.

5.2.2 **Second-level muon trigger**

As for calorimeter triggers, there is a large choice of commercial processors which might form the basis of a second-level muon trigger system. Alternative solutions include neural networks, associative memories and data-driven processors. The possibility of using information from the first-level trigger about the location of muon candidates is also valid here. However, the external muon chambers are expected to have very low occupancy, so there is much less to be gained from selective movement of data than in the case of calorimetry. Various architectures are possible for second-level muon triggering such as processor farms with each processor handling a different event, or architectures using a massively parallel computer system. The latter possibility has been studied [43]; using a system of associative string processors execution times of $\sim 10$ $\mu$s should be possible without making any reference to information from the first-level trigger.

5.3 **Triggers based on other detectors**

The rejection against background required at the first level is expected to be provided by triggers based only on calorimeters and external muon chambers. A number of other detectors, particularly those associated with electron identification, can help to provide additional rejection at the second level. We illustrate this with two examples: a preshower/tracker detector [44] and a transition-radiation detector [45].

5.3.1 **Preshower/tracker**

A preshower/tracker detector described in ref. [44] can be used at the trigger level to select electrons. The detector consists of several layers of silicon tracking, then a converter followed by several more layers of silicon tracking. Electrons are signalled by a single ionizing track before the converter which starts to shower before the second-tracking stage. Charged hadrons are unlikely to interact in the
converter, while electron pairs from converted photons (in a non-magnetic
detector) will give a twice-ionizing signal in the tracking before the converter.
Good rejection against background is obtained by exploiting the excellent
granularity of the detector when forming a coincidence between the silicon layers
before and after the converter, and with a high-energy cluster in the calorimeter.

It is apparent that the preshower/tracker trigger only makes sense when
combined with information from the calorimeter. Although the fast response of
the silicon detectors is compatible with the first-level time scale, the complication
of the interconnections necessary for a hard-wired trigger is likely to inhibit such
an option, leaving it for level 2. Combining information from different detectors
will require a common effort on trigger design. It is worth noting that the local-
global architecture presented in sect. 5.1.2 for a second level calorimeter trigger is
equally valid for any pad detector and in particular for a Silicon Tracker and
Preshower. The local processor array mapped onto the calorimeter can be mapped
at the same time onto the corresponding parts of the pad detector in front of the
calorimeter, in such a way that the association track-preshower-energy cluster can
be effectively done locally.

5.3.2 Transition radiation detector (TRD)

A TRD has the capability to separate electron and hadron tracks on the basis
of pulse height. A detector configuration suitable for the integration in a LHC
experiment [45] produces transition radiation X-rays in a foam radiator and detects
them using straw tubes. Digitization is performed using only three thresholds,
yielding 2-bit values for each channel. Roughly speaking, the first threshold is set
to be sensitive to a single-pion track, the second one to be sensitive to two
overlapping pion tracks, and the third one to be sensitive to an electron track.
Good separation of electrons is obtained by comparing the number of straws
above the third threshold with the number between the first two thresholds. A
possible second-level trigger architecture, which makes use of a massively parallel
computer system to find the tracks, is presently under study.

5.4 Higher level trigger

It is outside the scope of these lectures to cover after the second level trigger.
I will simply add a few comments on the major issues after level 2: the event
readout and assembly and the level 3 trigger proper.

5.4.1 Readout and event building

The first one is the readout of all the data from the local memories.
Although many factors contribute to the deadtime of an experiment, the readout
has always been the dominating one. At the LHC and SSC experiments the
readout time is going to be even more important with respect to the time between
triggers. Proper techniques for event readout, like event builders, have to be
adopted if we are to minimise deadtime. Event builders have evolved in the
latest generation experiments from simple single bus data collectors to parallel
multi-channels transporting the data to processor farms. Given the trigger rates
and the data volumes expected at the LHC and SSC experiments, fully parallel
event builders, free of inherent bottlenecks, have to be foreseen. Pre-event builder
buffering and fast data switches will be part of the system.
The critical role of event building is further illustrated by the considerations on the different levels of technological evolution. While the use of VLSI front-end circuitry has increased, in the past decade, by at least three orders of magnitude the allowable trigger rates, while the performance of high-level processors, RISC in particular, and the density of data storage have improved at an equivalent rate, the speed of standard busses normally used for event building has only improved by a factor of ten on the same time interval. Event building has become the bottleneck. Many options are being studied both with simulation tools and with prototype hardware.

5.4.2 Level 3 trigger and processor farm

Higher level algorithms will have to provide information about rejection factors of both number of events and overall data volume. That will only become clear after a good knowledge of the detector configuration, of the nature of the detector data, of the physics channels addressed and of the rejection already obtained at lower levels. It seems however very probable that algorithms close to final physics analysis will be necessary to achieve extra rejection, i.e. the "traditional" distinction between online and offline analysis will presumably be out of context in LHC/SSC experiments.

From the knowledge of the event rate, data volume and physics algorithms one will evaluate the computing power needs and study options of processor types and configurations to provide it. The farm architecture is, anyway, strongly linked to the event builder and data link configurations. Similarly to the case described in sect. 5.1.2, this issue must be at first a topic of simulation. Commercially available general purpose processors (e.g. RISC) should be the preferred option at this level as the most direct way of guaranteeing uniformity of the software environment (backend, offline).

6. FRONTEND READOUT

Undoubtedly, one of the most difficult technological challenge for LHC/SSC experimentation is in the area of front-end electronics. Although good progress is being made in many areas, the state-of-the-art of electronics is still inadequate in some aspects and it is widely recognized that a big R&D effort is necessary in order to meet the LHC time scale. In particular, industrial developments must be watched closely in order to take full advantage of the fast progress in this domain. For a good description of frontend elements features for LHC/SSC experimentation see ref. [46].

6.1. Frontend elements

A number of basic components, shown schematically in fig. 9, will be essential elements of any trigger and data acquisition system for the LHC and the SSC. Fast low-noise preamplifiers/shapers, Analogue-to-Digital Converters (ADCs), analogue and digital memories for pipelined readout structures, first-level trigger processors, data-flow processor systems and Digital Signal Processors (DSPs) will all have to be developed.
Fig. 9. Frontend components.

The demands of LHC include:

- very high performances for front-end applications:
  - 150 MHz bandwidth, Equivalent
  - Noise Charge of 400 electrons
  - at least 16 bit dynamic range and at least 10 bit resolution (for calorimeters)
- low-power dissipation (< 0.1 W), and compact packaging and cabling where required
- high reliability for detector electronics in closed areas
- radiation hardness (1–10 Mrad) for electronics mounted on the detectors (up to hundreds of MRad in some cases).

Prototypes exist which satisfy one or the other of the above requirements, but no one satisfies all of them. Extrapolating from the expected evolution of the technology trend (see sect. 6.4), we can anticipate that working solutions can be found in the next decade. Key issues are ASIC (Application Specific Integrated Circuits), CMOS (Complementary Metal Oxide Semiconductor), SOI (Silicon-On-Insulator), Bipolar-CMOS and Gallium Arsenide technologies. An extensive use of VLSI (Very Large System Integration) techniques will complete the picture.

6.1.1. Analog-to-digital conversion

Critical requirements come from calorimetry, for which a dynamic range equivalent to at least 16 bits and an accuracy equivalent to at least 10 bits are needed. Low-power consumption and radiation hardness will be critical issues for inner detectors, including an electromagnetic calorimeter, if digitization is done locally. Speed is critical for every detector in which digitization is done before the first-level trigger. Many techniques of conversion are available today. Alternatives to the standard flash ADC are the multistage flash ADC [47], ADCs based on the Σ–Δ technique [48] and pipelined ADCs [49]. Comparative analyses of
ADCs presently under development [50] show the relative advantages and
disadvantages, either in reality or in the promises of their development trends. In
any case, progress in this field is very fast based on new technology. The solution
adopted will depend on the detector type and the front-end electronics (e.g. at
which stage in the trigger hierarchy the digitization takes place).

6.1.2. Pipeline memories

As shown in fig. 6, the first-level trigger decision time will be much longer
than the bunch crossing period of 15 ns. The information from all the detector
channels must therefore be buffered until the first-level decision is available,
probably for ~2 µs. Given the high-data flow, the simplest organisation of the
buffering at this level is by means of pipelines with steps of (submultiples of)
15 ns and a fixed length equivalent to the decision time of the first-level trigger.

More will be discussed in sect. 6.2 about schemes based on analogue or on
digital pipelines. As a basic consideration, one can say that analogue delays have
an overall lower power consumption than digital ones and the number of signal
lines is smaller thus implying a higher level of packaging.

Several analogue pipelines have been successfully tested, with sampling
frequencies between 70 and 140 MHz, number of cells from 64 to 256, sequential or
random access and very low power consumption (~1 MW/channel) [51].
Alternative forms of analogue delays are the Surface Acoustic Waves (SAW)
circuits.

Digital delays can be implemented with FIFO (First-In-First-Out) memories,
to which one can directly couple circuits for data scaling, baseline cancellation or
filtering functions as part of the pipeline (see for example ref. [33]). An interesting
alternative approach is the use of a micropipeline architecture, based on a FIFO-
like structure which also provides a number of arithmetic elements, which can be
considered as intelligent memories, where the data are treated and transformed
while waiting for external events (in this case a trigger decision) [52].

6.1.3. Digital signal processing

The signals generated by a LHC calorimeter (or other detector) may extend
over more than one bunch crossing. In such cases, a digital channel with
programmable filter capability can be used to extract the physical information and
to associate events to a given bunch crossing. For detectors where particle
identification can be done by signal shape analysis in single channels, this can be a
part of the trigger process. Ignoring considerations of power consumption, density
of packaging and radiation hardness, it is very attractive to go digital at an early
stage, complementing the analogue shaping with a pipelined digital signal
processor. If prompt digitization is not possible, digital signal processing can still
be done after the first-level trigger, probably forming part of the second-level
trigger, and fulfilling a data compression role.

6.1.4. Optoelectronic signal transfer

Given the rather heavy requirements on data volumes and radiation
hardness in the frontend data flow, efficient alternatives could be provided by
analogue data transfer on optical fibres [53]. Detector electrical analogue signals are
translated into optical signals by electro-optic intensity modulators and
transferred via optical fibres to photodetector receivers on remote readout
stations. Compared to copper cables, this technique has the advantages of high
speed, very low power dissipation, compactness and freedom from electromagnetic interference. Good radiation hardness and adequate linearity and dynamic range make them very interesting candidates not only for the readout of central tracking detectors, but also for improved prompt triggering schemes.

6.2 Frontend configurations

Several architectures could be envisaged depending on the pulse characteristics and on the occupancy of the detector channel. Fig. 10 shows schemes of three different schemes with somewhat complementary fields of application.

![Diagram](image)

Fig. 10. Frontend configurations

Configurations with analogue pipelines may be preferred where high-packaging and low-power dissipation are critical issues, but they might be restricted to limited dynamic range and their timing and calibration are likely to be complex. Digital pipelines are simpler to control and offer large dynamic range, but their use might be limited to areas where high-power dissipation is acceptable, allowing fast digitization. A third type consists of a mixture of analogue and digital electronics, namely a one-stage analogue memory to store the pulse, coupled to a one bit shift register to identify the bunch crossing to which the content of the analogue memory belongs. The latter scheme is applicable to detectors for which the occupancy is such that the probability of having two hits in the same cell during the trigger decision time is negligible.

Despite the obvious advantages in design, commissioning and maintenance of having a unique scheme for all the detectors, that might not be possible given the very different nature of the detector signals and the extremely advanced technology required. Fig. 12 a comparative presentation of the main elements and overall features of configurations where the digitisation is performed before or after the signal pipeline.
6.3 Timing and system aspects

The synchronisation of the many million electronics channels, scattered over several tens of squared meters, to an accuracy of a nanosecond (or a fraction of, in some cases) is going to be one of the most delicate piece of the trigger/DAQ system. The problems that will have to be faced are far from being under control and will require very attentive studies of all the aspects concerned.

One can envisage a system by which a master timing signal, synchronised to the machine radio-frequency, is centrally generated from a location close to the detector and automatically compensated locally for the different phase delays of the various detector parts. Novel techniques are being investigated [54] for the distribution of the timing signal, in which different kind of control information and event taggings might be modulated. A solid scheme for the initial calibration of the synchronisation system and of its continuous monitoring has to be studied and will have to include careful planning of the electronics board layout on the detectors, precalculated length of cables and fibres, fine adjustment of the clock phase at the chip level and planning of dedicated data taking periods.

The system aspects relevant to the correct bunch assignment of the data and to event identification involve also a scheme for channel and event tagging to be applied to data in the frontend pipelines. Error detection and error recovery will also be of primary importance for a proper event reconstruction and a suitable scheme is far from being evident today.
The timing problem is not relevant for the frontend readout electronics, but to the one of the level 1 trigger as well. The global level 1 trigger operation must be synchronised to the bunch crossing as the pipelined data are and must give its accept/reject signal at a constant delay from the corresponding interaction (fixed latency).

6.4 Technology trends

The technology of CMOS and BiCMOS is quickly moving towards its physical limit. Compared with what was available ten years ago there is an increase of at least two orders of magnitude in memory density (fig. 12(a)) and in microprocessor performance (fig. 12(b)).

![Graph showing CMOS memory density vs. Year](image1)

**Fig. 12(a).** Trends in density of logic in micro-electronics.

![Graph showing Number of transistors/IC vs. Year](image2)

**Fig. 12(b).** Trends in processing power per chip.

Provided such trends continue, the projected compactness seems to match the requirements for the readout of highly granular detectors, and the projected speed matches the needs of fast front-end electronics and trigger systems. These projections might even be overtaken by technology changes, such as the use of GaAs components or the introduction of new processor architectures (e.g. massive parallelism). Obviously, in order to take full advantage of the technology the high-energy physics community must develop the capability of exploiting this progress.

6.5 Micro-electronics

Present evaluations performed in the LHC and SSC experimental contexts indicate that the investment necessary for the electronics in a typical experiment would amount to as much as 30% of the several hundred million Swiss francs (or dollars) total cost. A good fraction of the electronics will need to be micro-electronics: in the front end where it will have to be mounted directly on detectors, in the trigger and data acquisition systems in order to meet the requirements of complexity and power consumption, and at the interconnection level to provide high density, reliability and low-unit cost.

Micro-electronics is, therefore, a crucial issue for LHC and SSC experiments. Digital micro-electronics developments will be increasingly more reusable and easily adapted to the changes of technology. In the near future, increasingly big libraries of macrocells will be available, development systems will become more
standard and they should be used more easily by the designer. In order to avoid
duplication of effort, a coordination of European activities is being organised.
European initiatives, such as EUROCHIP, will help this by providing universities
and laboratories with the proper tools for the training of a new generation of
engineers. By taking advantage of such programmes, high-energy physics
laboratories and institutions should coordinate their efforts in order to pursue the
development of the cost-effective, advanced technology necessary for LHC
experiments.

A very important issue for electronics developments is the need of radiation
hardness, at least in some areas of the detectors. Detailed evaluations [55],
supported by various measurements, predict radiation levels of 1 to 10 MRad for
the locations of the inner detector electronics (and even more in the very forward
regions). Needless to say that the extent of the radiation problem at the LHC must
not be underestimated. There are encouraging signs of industrial interest in the
development of rad-hard electronics, motivated not quite by industrial needs of
radiation resistance, but rather by the superior speed performance obtained with
some of the production processes of radiation-hard components. The task of the
high-energy physics community is to develop test systems to ascertain their
usefulness.

6.6 Industry spin-offs

One of the main driving forces for the rapid innovation and the fast
appearance on the market of technological developments is the program for the
production of a new television standard. In the last 20 years the needs related to
telecommunications and, particularly, to television have strongly contributed to
the large scale industrial exploitation and development of standard technologies
(CMOS, BiCMOS). The popular video market provides the justification for the
enormous financial investment needed. Together with people in other fields,
researchers in fundamental physics have made extensive use of this “spin-off”.
Flash ADCs, analogue memories, personal computers, helical-scan recording,
data-compression techniques and image-processing systems are all examples of
television industry spin-offs from which we can profit.

In the last few years the extensive intellectual and technological resources of
the European studio and consumer television industry, broadcasters, PTTs and
universities have been brought together in a EUREKA project, EU95, to meet a
new challenge: the development of a High Definition Television (HDTV) [56].
Designed to progress in an evolutionary way, the HDTV project represents a
tremendous effort of R&D in the field of the standard technology.

The HDTV specifications are in a similar range of the ones needed for LHC
experiments. Transmission encoding of 144 MHz will require fast ADCs and high
speed, high-density mass storage devices; high-performance imaging will need
powerful image processors, digital filters, etc.; data communications of 1.3 Gbits/s
are also planned. Independently of the signal transmission technique, the signal
treatment at reception is fully performed digitally. Therefore, a TV channel might
be seen as an LHC detector channel, as pictorially shown if fig. 13, both from the
performance and from the functionality points of view: fast digitization,
enormous throughputs, inherent pipelining (one pixel after another instead of
one bunch crossing after the other).
Fig. 13. The HDTV chain compared to a readout channel of a LHC/SSC experiment.

In addition, the need to adapt multi-standard formats in a digital framework imposes the development of powerful programmable video-signal processors [57]. Their announced performance figures are very promising and the modular and programmable structure makes them ideal candidates for application in the first levels of LHC triggers. In conclusion, given the time scale defined for the EU95 project, analogue bandwidths of $\sim 100$ MHz and digital throughputs of $\sim 1$ Gbit/s can be expected in the middle of the 1990's as a spin-off from the European HDTV development. This does not mean that HDTV components could be used directly in LHC experiments. Other requirements do not match our needs: power consumption, density of packaging and radiation hardness are additional problems for the LHC. However, it is important to be aware that an extensive R&D program is under way in Europe addressing problems similar to ours. The use of HDTV prototypes to exercise architectures in realistic conditions and the combination of our expertise and competence with their methods and means would be beneficial for the solution of LHC problems.

7. R&D PROJECTS

As already mentioned, today's technology is inadequate for the realisation of a system capable of meeting the LHC/SSC requirements. Traditionally, the preparation of new generation experiments has concentrated on intense developments activities for detector techniques. For LHC/SSC, the need for a major research and development effort also in the field of electronics, triggering and data acquisition is being more and more recognised by the High Energy Physics community. As a consequence, several Research and Development projects have been recently initiated in the context of LHC, some in the context of detector prototype developments and others as independent R&D's. A detailed description of such activities is beyond the scope of this paper, but a list of the most relevant activities already gives a measure of the effort being undertaken by the LHC community. A complete list of references is to help the interested reader to find the full details of each project.
Projects related to Frontend Readout electronics:
- VLSI development of an analogue readout chip for a tracking/preshower detector.  
  Within DRDC project RD-02 [44]
- Front-end electronics for a transition radiation detector.  
  Within DRDC project RD-06 [45]
- Readout System Test Benches  
  DRDC project RD-12 [54]
- A digital front-end and readout microsystem for calorimetry at LHC.  
  DRDC project RD-16 [33]
- Front-end electronics for a Silicon pixel detector.  
  Within DRDC project RD-19 [58]
- Front-end electronics for a Silicon strip detector.  
  Within DRDC project RD-20 [59]
- Optoelectronic analogue signal transfers for LHC detectors.  
  DRDC proposal P31 [53]

Projects related to Level 1 Triggering:
- Study of muon triggers and momentum reconstruction in a strong magnetic field for a muon detector at LHC.  
  DRDC project RD-5 [14]
- Studies in fast calorimeter trigger design.  
  UK project [60]

Projects related to Level 2 Triggering:
- Embedded architectures for second-level triggering in LHC experiments.  
  DRDC project RD-12 [36]
- A network of Transputers as the basic structure for a Second and Third level trigger-processor farm.  
  NBI project [61]

Projects related to Level 3 Triggering and Data Acquisition:
- A Scalable data taking system at a test beam for LHC.  
  DRDC project RD-13 [62]
- HiPPI based Event Builder.  
  Joint project RD-11/RD-13 see above [61]
- A network of Transputers ...
- Applications of the scalable coherent interface to data acquisition at LHC.  
  DRDC proposal P33 [63]
- Simulation of system aspects.  
  Within DRDC project RD-13 [62].

8. CONCLUSIONS

The scenario presented in these lectures shows that building front-end electronics, triggers and data-acquisition systems for the LHC, although very difficult, it will not be impossible provided the ongoing R&D effort is continued, and in some areas enlarged, with sufficient investment of manpower and funding. The areas which are most in need of further study are those closest to the detectors where high speed is needed, in some cases in conjunction with low-power dissipation and radiation hardness. In particular, front-end analogue electronics, pipeline memories, fast ADCs, and first-level trigger processors need to be developed. In some of these areas, benefits might come from developments in industry which must be followed closely.

Further away from the detector (beyond the first-level trigger) more of the components of the trigger and data-acquisition system can come from industry.
High-performance processors, data links, networks and buses are already available and the technology is moving fast. Here we see a need to follow developments from industry by testing and evaluating the latest products. The challenge will be to combine the available technology to build the trigger and data-acquisition system. The complexity of such systems will require the use of modern software methods and software tools for design and modelling.

Differently of what done so far, requirements of the trigger and data-acquisition system in the overall detector design must be considered from the beginning of the design. Clearly, all detectors used in the first-level trigger must provide fast signals. In addition, physically large detectors will necessarily have a long first-level trigger decision time due to cable delays; it is important to make provision for short-cable runs from the outset. The choice of detectors may also affect the location of the front-end and first-level trigger electronics; problems of cabling must be weighed against problems of power dissipation and radiation hardness. Synchronization is a major issue that must not be neglected given the short-bunch crossing period. For every detector channel there will be pipeline and buffer memories, distributed over the detector, which must be synchronized to much better than 15 ns (the synchronization of movement of data from first-level pipelines to second-level buffers is particularly delicate). The message here is that the first-level readout electronics of all detectors and all the first-level trigger systems must be designed in a coherent way.

Considering the large event size and high first-level trigger rate, one is faced with very large data transfer rates. It is not desirable to move these data over large distances before second-level trigger processing unless there are constraints from the overall detector design. A related issue is that each trigger level can benefit from calculations performed at lower levels. For example, the first-level electron and muon trigger processors could be used to flag areas of the detector containing candidates to be validated by the second level. There is a general need for a coherent design of the trigger system both between different detectors and between trigger levels.

As a final remark, I hope that these lectures have managed to present the problem of signal processing, triggering and data acquisition at the LHC and SSC experiments as interesting enough to attract the attention of young physicists and engineers, challenging enough to motivate the participation of the brightest ones and not frightening to the point of discouraging any of you.

APPENDIX

This appendix briefly defines some of the most common High Energy Physics technical terms frequently used in this paper. It is only meant for those readers who are not familiar with the High Energy Physics jargon.

- **HEP**: High Energy Physics.
  
  Fundamental research aimed at disentangling the basic laws of nature by studying interactions between accelerated "elementary" particles.

- **Collider**: accelerator where particles collide head-on.

  Existing colliders: p–p, p–p̄, e+–e–, e––p
  
  Fundamental difference: electrons (leptons) are really elementary, therefore the whole beam energy goes in the collision
- protons (hadrons) are made of constituents (partons: quarks and gluons), carrying only a fraction of the particle energy and responsible of the basic interaction.

- **Event:** the result of an interaction between particles.

- **TeV:** particle energies are measured in electron-Volt (eV).
  
  Order of magnitude: today is GeV to TeV at supercolliders will be tens of TeV.

- **Cross section:** "probability" of interaction.
  
  It depends on the type of particle (for protons is higher than for electrons) and on the interaction energy (it increases for higher energies).
  
  Unit = barn (10⁻²⁴ cm²).

- **Luminosity:** "density" of interactions.
  
  It depends on the number of particles and the transverse size of the crossing beams.
  
  Unit = cm⁻² s⁻¹.
  
  Order of magnitude: today is 10³⁰ to 10³¹ at supercolliders will be 10³³ to 10³⁴.

- **Cross-section x Luminosity = Interaction rate**

**ACKNOWLEDGEMENTS**

I thank all the members of the working groups on Frontend Electronics, Triggering and Data Acquisition to which I have participated so far, for the amount of material made available and the stimulating discussions which deeply enlightened me. I also thank S. Cittolin and N Ellis for the very pleasant collaboration we had so far, and I still have, in organising and coordinating working groups in various occasions, especially the one of the Large Hadron Collider Workshop in Aachen. In particular, I want to thank them for letting me use the content of the proceedings and the transparencies which we have prepared together in occasion of the Aachen workshop.

**REFERENCES**


[39] Informal project, collaboration of members of RD11.[36] and RD16.[33].


[42] E. Petrollo et al., A fast tracking level-1 muon trigger for high-luminosity colliders using resistive plate chambers, Proc. of the Large Hadron


[56] EUREKA HDTV EU95, Bosch, Philips, Thomson et al. (19...).

[57] U. Schmidt et al., ITT Intermetall, Data-driven array processor for video signal processing, to appear in IEEE Transactions on Consumer Electronics 8/90 (1990);


[61] J.R. Hansen et al., A network of Transputers as the basic structure for a Second and Third level trigger-processor farm, NBI Project approved in Denmark.


An Introduction to Artificial Neural Networks

Carsten Peterson\(^1\) and Thorsteinn Rögnvaldsson\(^2\)

Department of Theoretical Physics, University of Lund, Sweden

Abstract

A general introduction to artificial neural networks is given assuming no previous knowledge in the field. Properties of the multilayer perceptron, feature maps, the Hopfield model and the Boltzmann machine are discussed in some detail. Also novel methods of finding good solutions of difficult optimization problems with feed-back networks and so-called elastic nets are described. Throughout the lectures practical hints on how to use the algorithms are given. Potential hardware implementations, both VLSI and optical, are briefly mentioned.

The power of the artificial neural network approach is illustrated in three high energy physics applications — quark jet tagging, mass reconstruction and track finding.

1 Introduction

There has been an upsurge in interest in Artificial Neural Networks (ANN) over the last 5 years. There are several reasons for this. One is that theoretical obstacles within the so-called perceptron, which was popular in the late sixties, have been overcome. Also the recently available inexpensive access to powerful and inexpensive CPU has facilitated the development of the field both with respect to model development, and very importantly, to deal with real-world problems. Indeed, results from impressive "product" quality application work keeps appearing both from the commercial and academic sector.

These lectures are intended as an introduction to the basic concepts of ANN for students in high energy physics (HEP). However, since the physics applications are presented in a separate section, they can be read by a more general audience. We focus mostly on architectures and algorithms relevant for HEP and have for example left out those specifically designed to handle speech recognition problems etc.. We refer to ref. [1] for a more comprehensive treatment of ANN. The lectures only assume very basic knowledge of ordinary differential equations, linear algebra (vector spaces, matrices, etc.) and statistical physics.

These lectures are organized as follows. Section 2 deals with neural systems in general, biological neural networks and the abstraction to artificial neural networks. Sections 3 to 6 describe the main ANN models and algorithms; supervised feed-forward networks, self-organizing networks, feed-back networks and networks used for optimization problems. In section 7 a very brief discussion on hardware implementations can be found. Section 8 describes some applications of these ANN algorithms to high energy physics problems and section 9 contains a short summary.

The reader who is merely interested in getting started with some HEP application can omit the Hopfield model and the Boltzmann machine (essentially section 5) with no loss in content. Also the lectures can be read coherently when ignoring the sections on variations and practical hints etc. in case the reader is only interested in the basic concepts.

2 Neural Networks

Artificial neural networks is a computational paradigm that differs substantially from those based on the "standard" von Neumann architecture. ANN's generally learn

\(^1\) thepcap@seldc52 (bitnet) carsten@theplu.se (internet)
\(^2\) thepdr@seldc52 (bitnet) demni@theplu.se (internet)
from "experience", rather than being explicitly "programmed" with rules like in conventional artificial intelligence (AI).

2.1 Biological Neural Networks

ANN is inspired from the structure of biological neural networks and their way of encoding and solving problems. We will here briefly review the basic components and functionality of the vertebrate central nervous system (CNS), whose details were revealed around 1940 with the emergence of the electron microscope. For more extensive literature on this subject we refer the reader to refs. [2, 3].

The human brain contains approximately $10^{12}$ neurons. These can be of many different types, but most of them have the same general structure (see fig. 2.1). The cell body or soma receives electric input signals to the dendrites by means of ions. The interior of the cell body is negatively charged against a surrounding extracellular fluid. Signals arriving at the dendrites depolarize the resting potential, enabling Na$^+$ ions to enter the cell through the membrane, resulting in an electric discharge from the neuron — the neuron "fires". The accumulated effect of several simultaneous signals arriving at the dendrites is usually approximately linearly additive whereas the resulting output is a strongly nonlinear all-or-none type process. The discharge propagates along the axon to a synaptic junction, where neurotransmitters travel across a synaptic cleft and reach the dendrites of the postsynaptic neuron. A synapse which repeatedly triggers the activation of a postsynaptic neuron will grow in strength; others will gradually weaken. This plasticity, which is known as the Hebb rule, plays a key part in learning.

![Synaptic Junction](image)

**Figure 2.1: Schematic structure of a neuron**

This general neuron structure is implemented in many different sizes and forms with different functionalities. Cell bodies have diameters in the range 5–80 $\mu$m and dendrite "trees" extend from 10 $\mu$m up to 2–3 mm. Axons can be up to 1 m in length, why sizes of entire neurons vary from 0.01 mm to 1 m. Primary sensory neurons connect muscles or receptors to neurons, secondary sensory neurons and interneurons connect neurons with neurons, while motor neurons connect neurons with muscle fibers.

The connectivity (number of neurons connected to a neuron) varies from a $O(1)$ to $O(10^5)$. For the cerebral cortex $O(10^3)$ is an average. This corresponds to $O(10^{15})$ synapses per brain. Synapses can be either excitatory or inhibitory of varying strength. In the simplified binary case of just two states per synapse the brain thus has $O(2^{10^{45}} \approx 10^{10^{44}})$ possible configurations! The neural network is consequently in sharp contrast to a von Neumann computer both with respect to architecture and functionality (see table 2.1).

The von Neumann computer was originally developed for "heavy duty" numerical computing but has later also turned out to be profitable for data handling, word processing and suchlike. However, when it comes to matching the vertebrate brain in terms of performing "human" tasks it has very strong limitations. There are therefore strong reasons to design an architecture and algorithm that shows more resemblance with that of the vertebrate brain.
Table 2.1: Comparison of characteristics of neural networks and conventional computers.

2.2 Artificial Neural Networks

The philosophy of the ANN approach is to abstract some key ingredients from biology and out of those construct simple mathematical models that exhibit most of the above-mentioned appealing features. In physics one has good experience of model building out of major abstractions. For example, details of individual atoms in a solid can be lumped into effective "spin" degrees of freedom in such a way that good description of collective phenomena (phase transitions etc.) are obtained. It is exactly the collective behavior of the neurons that is interesting from the point of view of intelligent data processing.

2.2.1 Basics

The basic computational entities of an ANN are the neurons [4] 𝑣𝑖, which can take real values within the interval [-1,1] (or [0,1]). Sometimes the even simpler binary neuron is used, where 𝑣𝑖 = {-1, 1} (or {0, 1}). For binary neurons the notation 𝑠𝑖 is normally used.

These are simplifications of the biological neurons described in the previous section. The most common form is the "thresholding" neuron (see fig. 2.2):

\[ 𝑣𝑖 = g(\sum \omega_{ij}𝑣j + \theta_i) \]  

(2.1)

where \(v_j\) are all neurons that are feeding to neuron \(v_i\), through weights (synapses) \(\omega_{ij}\). These weights can have both positive (excitatory) and negative (inhibitory) values. The \(\theta_i\) term is a threshold, corresponding to the membrane potential in a biological neuron. The non-linear transfer function \(g()\) is typically a sigmoid-shaped function like

\[ 𝑣𝑖 = g(𝑎𝑖) = \tanh(𝑎𝑖/𝑇) \]  

(2.2)

for [-1,1] neurons or

\[ 𝑣𝑖 = g(𝑎𝑖) = \frac{1}{2}[1 + \tanh(𝑎𝑖/𝑇)] \]  

(2.3)
for [0,1] neurons. The argument \( a_i \) is the linearly summed input signal to neuron \( v_i \):

\[
a_i = \sum_j \omega_{ij} v_j
\]  

(2.4)

and the "temperature" \( T \) sets the inverse gain of the function \( g() \); a low temperature corresponds to a very steep sigmoid and a high temperature corresponds to an approximately constant \( g() \) (see fig. 2.2). The limit \( (T \to 0) \) corresponds to binary neurons \( s_i \).

Figure 2.2: A thresholding neuron: (a) Neuron updating and (b) sigmoid response functions of eqs. (2,3) for different temperatures \( T \).

The simple "thresholding neuron" mimics the main features of real biological neurons in terms of linear additivity for the inputs and strong non-linearity for the resulting output. If the integrated input signal is larger than a certain threshold \( \theta_i \) the neuron will "fire".

There are two different kinds of architectures in neural network modeling; feed-forward (fig. 2.3a) and feed-back (fig. 2.3b). In feed-forward networks signals are processed from a set of input units in the bottom to output units in the top, layer by layer, using the local updating rule of eq. (2.1). In feed-back networks, on the other hand, the synapses are bidirectional. Activation continues until a fixed point has been reached, reminiscent of a statistical mechanics system. In our HEP applications feed-forward architectures will be used for pattern recognition, whereas track-finding will be based on feed-back architectures.

Figure 2.3: (a) Feed-forward and (b) feed-back architectures

2.2.2 Application Areas

How can the power of these networks be exploited? There are four major application areas with soft borderlines in between — feature recognition, function approximation, associative memories and optimization. Feature recognition is the most exploited application domain to date.
Figure 2.4: A one hidden layer feed-forward architecture.

Feature recognition

In feature recognition (or pattern classification) situations one wants to categorize a set of input patterns \( \mathbf{x}^{(p)} = (x_1, x_2, ..., x_N)^{(p)} \) in terms of different features \( o_i \). The input patterns are fed into an input layer (receptors) and the output nodes represent the features. For the architecture depicted in fig. 2.4, \( o_i \) depends on \( \mathbf{x}^{(p)} \) through the functional form (cf. eq. (2.1))

\[
o_i(\mathbf{x}^{(p)}) = g \left( \sum_{j=0}^{\omega_{ij}} g \left( \sum_{k=0}^{\omega_{jk}} x_k^{(p)} \right) \right)
\]

(2.5)

where \( \omega_{ij} \) and \( \omega_{jk} \) are the parameters. Eq. (2.5) can of course be generalized to any number of layers. The thresholds, or bias terms, \( \theta_i \) appearing in eq. (2.1) have in eq. (2.5) been generalized to weights \( \omega_{i0} \) by adding an extra "dummy" unit \( v_0 \) to each layer, which is constantly firing. Fitting \( \omega_{ij} \) and \( \omega_{jk} \) to a given data set (or learning) takes place with gradient descent on a suitable error function\(^3\). In this process the training patterns are presented over and over again with successive adjustments of the weights. Once this iterative learning has reached an acceptable level, in terms of a low error, the weights are frozen and the ANN is ready to be used on patterns it has never seen before. The capability of the network to correctly characterize these test patterns is called generalization performance. This procedure is equivalent to "normal" curve fitting where a smooth parameterization from a training set is used to interpolate between the data points (generalization).

Function approximation

Rather than having a "logical" output unit \( o_i \) with a threshold behavior described by eqs. (2.2,2.3), one could imagine having an output representing a real number, corresponding to replacing eqs. (2.2,2.3) with linear behaviors. In this case one adjusts the weight parameters to parameterize an unknown real-valued function. Such an approach can be useful in time-series predictions, where one aims at predicting future values of a series given previous values \( [5, 6, 7, 8] \), e.g.

\[
x_t = F(x_{t-1}, x_{t-2}, ...)
\]

(2.6)

where \( x_t \) is the real-valued output node and \( x_{t-1}, x_{t-2}, ... \) are the values of the series at previous times. Another application with linear output nodes is mass reconstruction in high energy physics \( [31] \), which is discussed in section 8.

Associative memory

In this kind of application there is no input/output distinction. Rather the network learns a set of identity mappings by fitting \( \omega_{ij} \) to memory patterns or "words" \( \mathbf{s} = \)

---

3) The error function is sometimes referred to as Lyapunov function or cost function.
(s₁, s₂, s₃, ..., s₉). If an incomplete or partly erroneous memory is presented to the network, it completes the pattern. In a certain sense feature recognizers are special cases of associative memories — pattern completion.

Optimization

Feed-back networks have shown great promise in finding good solution problems to difficult combinatorial optimization problems. In this case again non-linear updating equations (eq. (2.1)) are allowed to settle. This corresponds to minimizing an energy function, which typically looks like

\[ E = -\frac{1}{2} \sum_{ij} \omega_{ij} s_is_j \]  \hspace{1cm} (2.7)

The problem is mapped onto the form of eq. (2.7) by a clever choice of \( \omega_{ij} \). In this case \( \omega_{ij} \) are fixed once and for all for each problem — they are not adaptive as in the other domains of application. The neurons \( s_i \) are then allowed to settle into a stable state, where the solution to the problem is given by the configuration \( \tilde{s} = (s_1, s_2, ...) \) with minimum energy.

3 Feed-forward Networks

In this section we discuss feed-forward networks with respect to what mappings they can perform for a given complexity of the architecture. Also, different error measures and the back-propagation learning algorithm with variants and refinements are explained.

3.1 The Simple Perceptron

The simple perceptron [9], which dates back to 1961, has one layer of input units and one layer of output (or feature) units (see fig. 3.1). Starting from randomly distributed \( \omega_{ij} \)'s each pattern is processed through the network and the output nodes are updated according to eq. (2.1). For each training pattern \( \tilde{x}^{(p)} \) there is a known target pattern \( \tilde{t}^{(p)} \). The goal is to produce a true mapping \( \tilde{x}^{(p)} \rightarrow \tilde{t}^{(p)} \) for all \( p \).

![Figure 3.1: The simple perceptron.](image)

3.1.1 A Perceptron Learning Algorithm

The training of the perceptron is performed by minimizing the summed square error function

\[ E = \frac{1}{2} \sum_p \sum_i (s_i^{(p)} - t_i^{(p)})^2 \]  \hspace{1cm} (3.1)

with respect to the weights. This is done with a gradient descent method where the weights are incrementally updated in proportion to \( \partial E/\partial \omega_{ij} \). In practice this can be done either in batch-mode (or off-line mode) where \( \partial E/\partial \omega_{ij} \) are computed for all patterns before updating the weights or with an on-line procedure, where \( \partial E/\partial \omega_{ij} \) is computed pattern by pattern. In what follows we assume the latter alternative (in order to suppress the index \( p \)). The derivative of \( E \) with respect to \( \omega_{ij} \) is given by

\[ \frac{\partial E}{\partial \omega_{ij}} = \frac{\partial E}{\partial s_i} \frac{\partial s_i}{\partial \omega_{ij}} \]  \hspace{1cm} (3.2)
where $o_i$ is (cf. eq. 2.1)

$$o_i = g(\sum_{j=0}^{\omega_{ij}x_j}) = g(\bar{\omega}_i \cdot \bar{x})$$  \hspace{1cm} (3.3)

Gradient descent (or forward Euler) updating then reads

$$\Delta \omega_{ij} = -\eta \frac{\partial E}{\partial \omega_{ij}} = -\eta \delta_i x_j$$  \hspace{1cm} (3.4)

where $\eta$ is the learning rate parameter ($\eta < 1$) and $\delta_i$ is given by

$$\delta_i = (o_i - t_i)g'(\bar{\omega}_i \cdot \bar{x})$$  \hspace{1cm} (3.5)

When no more changes (within some accuracy) occurs, i.e. $\Delta \omega_{ij} \approx 0$, the weights are frozen and the network is ready to use for data it has never "seen". The procedure is summarized in fig. 3.2.

Note that this algorithm is not the only training algorithm for perceptrons (for other alternatives see refs. [10, 11]).

1. Initialize $\omega_{ij}$ with $\pm$ random values.
2. Repeat until $\omega_{ij}(t + 1) \approx \omega_{ij}(t)$:
   2.1 Pick pattern $p$ from training set.
   2.2 Feed input $\bar{x}^{(p)}$ to network and calculate the output $\bar{o}$ (eq. (3.3)).
   2.3 Update the weights according to
       $$\omega_{ij}(t + 1) = \omega_{ij}(t) - \eta \delta_i x_j$$
       where $\delta_i$ is given by eq. (3.5).

Figure 3.2: A perceptron learning algorithm.

### 3.1.2 Examples: Boolean Functions

To demonstrate the computational capabilities of the simple perceptron, let us consider the case of boolean functions of two binary inputs. Two such functions, the AND and OR functions, are defined in table 3.1.

<table>
<thead>
<tr>
<th></th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$t$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.1: The AND and OR functions.

Both the AND and OR functions are easily learned by the simple perceptron. Using a network with two inputs and an output unit with a $[0,1]$ transfer function $g()$ (eq. (2.3)) one might get the weight values $\bar{\omega} = (\omega_0, \omega_1, \omega_2) = (-1.5, 1.0, 1.0)$ for the AND problem and $\bar{\omega} = (-0.5, 1.0, 1.0)$ for the OR problem, where $\omega_0$ corresponds to the threshold term $\theta$ in eq. (2.1). It is easily verified that these weight values really compute the logical functions in table 3.1, by computing the argument $\bar{\omega} \cdot \bar{x}$ and using the step-function limit of $g()$ ($T \to 0$) for the output$^4$.

$^4$ The threshold "dummy" unit $x_0$ for the input layer is a constant $= 1$. 

3.1.3 Limitations

What are the limitations of the simple perceptron? Let us examine how it deals with the boolean exclusive-or [XOR], or the two-bit parity problem, which is defined in table 3.2. We again use a network with two inputs and one output node. To simplify the discussion we use the step-function limit of \( g() \). The functional dependence reads:

\[
o = g(\overline{\omega} \cdot \overline{z}) = \begin{cases} 
0 & \text{if } \omega_0 + \omega_1 x_1 + \omega_2 x_2 < 0 \\
1 & \text{if } \omega_0 + \omega_1 x_1 + \omega_2 x_2 > 0
\end{cases}
\]  

(3.6)

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(t)</th>
<th>(\overline{\omega} \cdot \overline{z})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(\omega_0 + \omega_1 + \omega_2 &lt; 0)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(\omega_0 + \omega_1 &gt; 0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(\omega_0 + \omega_2 &gt; 0)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(\omega_0 &lt; 0)</td>
</tr>
</tbody>
</table>

Table 3.2: The XOR problem and the conditions on weights for the simple perceptron.

It is clear that the inequalities in the right column of table 3.2 cannot be simultaneously satisfied. The three parameters are not sufficient to describe the problem. Another way to phrase this fact is that the XOR problem is not linearly separable, which is illustrated in fig. 3.3.

![Figure 3.3: State space of the XOR problem.](image)

The input space is two-dimensional and the possible inputs are corners of a square. The two "classes" \( t = 1 \) and \( t = 0 \) cannot be separated with a single line in the input space. For the linearly separable AND and OR functions, the separating line between the classes is orthogonal to the vectors \( \overline{\omega} = (\omega_1, \omega_2) \) for the two functions, which is easily verified, and the threshold term \( \omega_0 \) is a measure of the perpendicular distance to the origin. In the general case, the simple perceptron is capable of classifying all classes that are separable by a hyperplane \((N-1)\)-dimensional subspace) in the \(N\)-dimensional input space.

The failure of the simple perceptron is thus caused by its linearity\(^5\) and this criticism \([12]\) against it was vigorously pursued in the late sixties, which effectively put ANN research on hold for some 20 years. One might argue that a XOR problem is too academic to serve as a testbed for an adaptive learning algorithm but many real-world pattern recognition problems contain similar higher order constraints which make them linearly inseparable. To circumvent this we have to provide the perceptron with non-linear qualities. One way to do this is of course to use a more complex transfer function \( g() \), like a Gaussian. In that case the simple perceptron would be able to compute the XOR problem. Another way, which is more powerful and general, is to introduce extra hidden layers between the input units and the output units, which is usually referred to as the multilayer perceptron (MLP).

\(^{5}\) The non-linearity of the sigmoid function \( g() \) is irrelevant for the functionality; it is only used to provide a feature representation.
3.2 The Multilayer Perceptron

To improve the simple perceptron we augment it with a layer of hidden units (see fig. 2.4). Since these have non-linear response functions (eq. (2.5)) the network should be able to perform non-linear mappings. The hidden nodes in fig. 2.4 build up an internal representation of the data. In fig. 3.4 we show an explicit solution to the XOR problem with one layer of two hidden nodes.

![Diagram of XOR problem solution](image)

Figure 3.4: (a) An explicit solution to the XOR problem. (b) The representation (hyperplanes) of the hidden units; $\vec{w}_1 = (0.5, -1, 1)$ and $\vec{w}_2 = (0.5, 1, -1)$. The output unit performs a NAND function on $h_1$ and $h_2$.

3.2.1 The Back-propagation Learning Algorithm

Let us now generalize the gradient descent procedure of eq. (3.4) to the case with hidden layers [13]. For simplicity we perform the derivations for architectures with one hidden layer - the generalization to several hidden layers is straightforward. At each layer we introduce the notation $a_i$ and $a_j$ for the weighted input sums (the units $h_0$ and $x_0$ correspond to the threshold “dummy” units).

\[
a_i = \sum_{j=0}^{\infty} \omega_{ij} h_j = \vec{w}_i \cdot \vec{h}
\]

(3.7)

\[
a_j = \sum_{k=0}^{\infty} \omega_{jk} x_k = \vec{w}_j \cdot \vec{x}
\]

(3.8)

The hidden and output nodes are thresholded like

\[
h_j = g(a_j/T)
\]

\[
a_i = g(a_i/T)
\]

(3.9)

In addition to $\partial E / \partial \omega_{ij}$ we now also need $\partial E / \partial \omega_{jk}$ in order to minimize $E$. With the "chain-rule" one gets

\[
\frac{\partial E}{\partial \omega_{jk}} = \sum_i \frac{\partial E}{\partial a_i} \frac{\partial a_i}{\partial h_j} \frac{\partial h_j}{\partial a_j} \frac{\partial a_j}{\partial x_k} = \sum_i \omega_{ij} \delta_i g'(a_j) x_k
\]

(3.10)

where $\delta_i$ is defined in eq. (3.5).

With gradient descent, the $\omega_{ij}$'s are again updated as in eq. (3.4). For $\omega_{jk}$ one gets

\[
\Delta \omega_{jk} = -\eta \sum_i \omega_{ij} \delta_i g'(a_j) x_k
\]

(3.11)

Here we see how the error $\delta_i$ is back-propagated down through the network.
1. Initialize $w_{ij}$ with $\pm$ random values.
2. Repeat until $w_{ij}$ and $w_{jk}$ have converged
or the desired performance level is achieved:
   2.1 Pick pattern $p$ from training set.
   2.2 Present input $x^{(p)}$ and calculate $\tilde{h}$ and
   the output $\tilde{o}$ according to eq. (3.9)
   2.3 Update the weights according to
   \[
   \begin{align*}
   w_{ij}(t + 1) &= w_{ij}(t) - \eta \delta_i h_j \\
   w_{jk}(t + 1) &= w_{jk}(t) - \eta \sum_i \omega_{ij} \delta_i g'(a_j)x_k
   \end{align*}
   \]
   (...etc...for extra hidden layers).

Figure 3.5: The back-propagation learning algorithm.

Back-propagation is not limited to the simple case of layer to next-layer connections, it generalizes just as easily to network architectures with input to output connections besides the input to hidden and hidden to output.

The training of a multilayer perceptron using BP can be thought of as a walk in "weight space" along an energy surface, defined by eq. (3.1), trying to find the global minimum and avoiding local minima. There is, in contrast to the simple perceptron, no guaranty that the global minimum will be reached with this method. However, in most cases the energy landscape is smooth without too many local minima.

3.2.2 Refinements and Variations

One major criticism against this "vanilla" version of BP is that the convergence time grows very rapidly with the problem size. Several variations have been suggested to speed it up [14]. One straightforward improvement is to use "on-line" updating (see above) where the weights are updated for each $n^{th}$ pattern presentation (where $n \sim O(10)$) instead of for the whole training set [15] (for some problems [16], however, "off-line" updating has been reported to be quicker). In this section we describe some useful variations and extensions to the back-propagation algorithm in order to make it more powerful.

Alternative error measures

The summed square error in eq. (3.1) is not the only possible error measure. Any function that is differentiable with respect to $a_i$ and has a minimum for $a_i = t_i$ can be used. Some commonly used alternative measures are:

- Cross-entropy error: [17]
  \[
  E = - \sum_p \sum_i [t_i^{(p)} \log a_i + (1 - t_i^{(p)}) \log(1 - a_i)]
  \]  
  (3.12)

In this case the $g'$ factor in the updating of $w_{ij}$ disappears, but the updating of the hidden-hidden and input-hidden connections are the same (eq. (3.10)).

- Kullback measure: [18]
  \[
  E = \sum_p \sum_i t_i^{(p)} \log \frac{t_i^{(p)}}{a_i}
  \]  
  (3.13)

which is used in conjunction with $K$-valued sigmoids (see below). The back-propagation algorithm turns out to be the same as in the case of the entropy error above, again without a $g'$ between top hidden and output layers.
Alternative output nodes

It is sometimes convenient to use other output nodes than the normal “thresholding” types in eqs. (2.2) and (2.3).

- **K-valued Potts neurons:** If the target output vectors $\vec{t}^{(p)}$ equal the orthonormal basis vectors $(1,0,0,...)$, $(0,1,0,...)$, $(0,0,1,...)$, etc., then a “winner-takes-all” interpretation of the output $\vec{a}^{(p)}$ is often beneficial. An efficient encoding of this is achieved by using $K$-valued neurons, analogous to Potts spin models in physics [19]. The sigmoid $g()$ for the output node is then replaced by

$$o_i = o_i(a_1, a_2, ..., a_n) = \frac{e^{a_i/T}}{\sum_i e^{a_i/T}}$$

(3.14)

satisfying

$$\sum_i o_i = 1$$

(3.15)

Eq. (3.15) corresponds to a dimensional reduction of the solution space hypercube to a hyperplane (see fig. 3.6).

![Diagram](image)

Figure 3.6: The volume of solutions corresponding to $K$-valued neurons for $K=3$. The shaded plane corresponds to eq. (3.15) for $K=3$.

- **Linear output nodes:** If one wants to use the network as a plain “fitting engine”, like in the case of time-series prediction, the output nodes should encode any real number; i.e. be linear $o_i = a_i$. In this case $g()$ also disappears from eqs. (3.2) but the lower layers are the same.

Adapting parameters

The back-propagation in its basic form has two parameters; the temperature (inverse gain) $T$ and the learning rate $\eta$. During training, the network passes through different parts of the energy landscape, with possibly very different profiles. The learning parameters should therefore be allowed to change dynamically during training [1]. The ideal is to let each parameter change individually for each weight [20].

- **Annealing:** In the high-temperature limit ($T \to \infty$), the sigmoid $g(x)$ approaches a constant value. For large temperatures, the energy is thus approximately constant, independent of the weights. Slowly decreasing the temperature will increase the contours of the energy landscape, initially with small gradients, making it easier to find an appropriate learning rate. This is usually not necessary.

- **Momentum:** In situations, where the energy landscape is characterized by steep walls and “flat” valleys, it is beneficial to average over many updatings in order to quickly reach the bottom of the valley and increase speed along it. The most commonly used technique to do this is the addition of a momentum term. The idea is to provide each weight with some inertia in order to avoid oscillations. This is done
by including an extra term in eqs. (3.4,3.11)

\[ \Delta \omega(t + 1) = -\eta \frac{\partial E}{\partial \omega} + \alpha \Delta \omega(t) \] (3.16)

where \( \Delta \omega(t) \) refers to the previous updating. The momentum parameter \( \alpha \) can be varied during training.

- **Second order methods**: A more thorough approach to dynamically tune \( \eta \) with respect to the landscape is the *Newton rule*:

\[ \Delta \omega_{ij} = -\eta H^{-1} \frac{\partial E}{\partial \omega_{ij}} \] (3.17)

where the Hessian matrix \( H \) is given by

\[ H = \frac{\partial^2 E}{\partial \omega_{ij} \partial \omega_{ij}} \] (3.18)

Needless to say this procedure is too CPU and memory consuming to be of practical use; for \( N \) weights it requires the storage and diagonalization of an \( N \) by \( N \) matrix. Also, for networks with many hidden layers, higher order methods sometimes perform worse than simple gradient descent [21].

**Manhattan Learning**

The time-dependence of the learning rate for the standard BP algorithm sometimes makes it difficult to reach optimum performance. In such cases *Manhattan updating* can be more efficient. Manhattan updating is bounded and uses only the direction of the gradient \( \nabla E \);

\[ \Delta \omega_{ij} = -\eta \cdot \text{sgn}\left(\frac{\partial E}{\partial \omega_{ij}}\right) \] (3.19)

where \( \eta \) should be decreased with learning. In this way the same learning “step” is used for all weights.

**Restricting the Number of Parameters**

An important question is what ANN architecture (number of layers, hidden nodes and degree of connectivity) to use for a particular problem. Clearly one should use as few parameters (weights) as possible in order to have powerful generalization properties. In other words one wants to avoid overfitting. A straightforward, but costly, way is of course trial-and-error. However, more algorithmic methods exist:

- **Weight decay**: The most simple such approach is weight decay, where weights which are rarely updated are allowed to decay according to

\[ \Delta \omega_{ij} = -\eta \frac{\partial E}{\partial \omega_{ij}} - \varepsilon \omega_{ij} \] (3.20)

where \( \varepsilon \) is the decay parameter, typically a very small number, \( \mathcal{O}(10^{-4}) \). This corresponds to adding an extra complexity term to the energy function

\[ E \rightarrow E + \frac{\varepsilon}{2\eta} \sum_{ij} \omega_{ij}^2 \] (3.21)

imposing a “cost” for large weights.
Pruning: A more advanced complexity term [8] is

$$E \to E_0 + \lambda \sum_{ij} \frac{\omega_{ij}^2/\omega_0^2}{1 + \omega_{ij}^2/\omega_0^2}$$ \hspace{1cm} (3.22)

where the sum extends over all weights. For large $|\omega_{ij}|$, the cost is $\lambda$, whereas for small weights it is zero. The scale of the weights is set by $\omega_0$. In this way the cost reflects the number of weights, instead of the size, hence the network gets pruned to only contain weights that are really needed to represent the problem.

Self-generating networks: Instead of determining the optimal architecture for a network by starting with a large number of weights and then use the pruning procedure described above, one could allow the network to change its complete architecture during learning. Such self-generating networks [22, 23, 24, 25] aim at constructing the most economic architecture for a specific problem by some algorithmic method. These generative algorithms generally perform better than simple BP, both in sense of convergence time and generalization.

Stochastic Methods

There exists a number of methods with stochastic elements aimed at avoiding getting stuck in local minima in the energy landscape. These include randomly changing the order of pattern presentations, to avoid cyclic behavior.

- Noisy input: Adding noise to the inputs smears the inputs and improves generalization. Too much noise however deteriorates the network’s performance.

- Random changes of weights: One can make stochastic steps increasing the energy by using e.g. the Metropolis [26] or heat bath [27] algorithms or the Langevin equations [28]. The latter case is natural since the gradient descent updating equations can be regarded as forward Euler approximations of the differential equations

$$\frac{d\omega_{ij}}{dt} = -\frac{\partial E}{\partial \omega_{ij}}$$ \hspace{1cm} (3.23)

with $\Delta t = \eta = 1$. Augmented with a white appropriately normalized Gaussian random noise $\sigma$ eq. (3.23) becomes the Langevin equation

$$\frac{d\omega_{ij}}{dt} = -\frac{\partial E}{\partial \omega_{ij}} + \sigma$$ \hspace{1cm} (3.24)

3.2.3 Practical Issues and “Rules of Thumb”

Even though we have a powerful learning algorithm for the multilayer perceptron, there are still a number of questions that remain unanswered. How many hidden layers and units are needed? How should a network be trained to achieve optimal generalization performance? What is “optimal generalization performance”? Some of these questions have simple answers while others don’t. We will in this section discuss these issues and give some rules of thumb for the questions with no firm answer.

Number of Layers

The “number of layers” issue of course depends on the specific task one wants the network to perform, but a general statement is that no more than two hidden layers are needed, even though very many units might be needed in these layers. The most common tasks for a BP feed-forward network are pattern classification (or feature recognition) and function approximation, which are discussed below.

- Pattern classification: Pattern classification means selecting regions of the input space and assigning them to classes. This is illustrated in fig. 3.7, where three classes occupy a two-dimensional input space. To separate these classes, the network has to make
borders around them and "cut them out". To see how this can be done we make use of the vector representation of eqs. (3.7) and (3.9). In analogy with the simple perceptron the units \( h_j^{(1)} \) in a first hidden layer are able to linearly separate the input space into two regions. The units \( h_j^{(2)} \) in a second hidden layer, corresponding to the output layer of fig. 2.4, can perform AND or OR like functions on \( h_j^{(1)} (h_j^{(1)} + h_j^{(1)}, h_j^{(1)} - h_j^{(1)}, h_j^{(1)} \cap h_j^{(1)}, h_j^{(1)} \cup h_j^{(1)}) \). The response region (in input space) of the units \( h_j^{(2)} \) will thus be a convex subspace. Analogously the output units in such a four-layered MLP will respond to AND or OR like functions on convex subspaces. Such subspaces can be of any shape, even concave. Subsequently, any classification task can be performed using only two hidden layers. In the case depicted in fig. 3.7 one hidden layer is not enough to give a simple classification, class B needs two hidden layers to be optimally classified.

**Function approximation:** [5] An arbitrary function \( F \) can be approximated by a linear combination of "splines" (a spline is a Gaussian-like bump). A spline-like response can be constructed by combining two layers, i.e. multiplying two sigmoids give a bump. This means that at most two layers are needed to approximate any function. Actually, it has been shown that any continuous function can be approximated with only one layer of sigmoidal units [29].

![Figure 3.7: A pattern classification problem in two dimensions.](image)

**Number of Hidden Units**

The approximate minimum number of hidden units to achieve a certain task can be determined from general arguments. It is however not wise to use the very minimum number of hidden units, since the number of paths to the global minima increases with the number of hidden units, decreasing the chance of getting stuck in a local minima. The addition of extra weights (= extra dimensions in weight space) adds the possibility of "going around the mountain instead of over it". But at the same time, more hidden units can give rise to lower generalization performance (see below). The ideal approach is to train with many hidden units and then decrease that number as the network is getting close to the global minima [25] (cf. the discussion on pruning above).

**Pattern classification:** The view of pattern classification as a "cut out" procedure gives a clue to the minimum number of hidden units needed for a specific classification task. In the "hard" case where the first hidden layer has to define the border of a closed volume in the input-space, at least \( (N + 1) \) hidden units are needed in the first layer for each class, usually more. This is because \( (N + 1) \) is the minimum number of hyperplanes needed to cut out a closed \( N \)-dimensional volume. The more hyperplanes one uses, the smoother the border of the class. The second hidden layer is usually performing some kind of AND or OR function on the units of the first hidden layer, why a lower number of units is needed there (compare with the simple perceptron
above).

- **Function approximation**: The number of hidden units needed to approximate a given function $\mathcal{F}$ is related to how many terms that are needed in an expansion of $\mathcal{F}$ in the function $g()$ [32].

**Learning Parameters**

The crucial parameter in BP is the learning rate $\eta$. The ideal $\eta$ depends on two things; the scale of the activations and the “fan-in” of the network.

- **Activation scale**: If the average activation $\langle z_i \rangle$ of input node $i$ is large, the optimal $\eta$ for weights connecting to that unit will be small. The natural thing is to rescale the input data such that $\langle z_i \rangle \sim \mathcal{O}(1)$ for all $i$, in which case they will all have approximately the same optimal $\eta$ (even the thresholds). This also simplifies for the case of the hidden units, where $\langle h_j \rangle \sim \mathcal{O}(1)$ for sigmoidal units, and the same learning rate can be used for all weight layers (if the “fan-in” is the same).

- **“Fan-in”**: The “fan-in” to a unit is the number of units connecting to that unit, in the forward direction. For example, if a fully connected three-layered (one hidden layer) MLP has 20 input units, 10 hidden units and one output unit, then the “fan-in” to a unit in the hidden layer is 20 and to the output unit it is 10. The optimal learning rate for connections $\omega_{jk}$ scales like $\eta \propto 1/("fan-in" \text{ to unit } h_j)$ or stronger [30].

The momentum term $\alpha$ controls the “averaging” of the updatings and is closely connected to the learning rate. An increase in $\alpha$ means an increase of the “effective” learning rate. The optimum $\alpha$ depends on the updating procedure used. For the “off-line” method $\alpha$ is very useful and should be a number close to unity ($0.5 < \alpha < 1$). For the “on-line” updating, on the other hand, $\alpha$ is often (but not always) useless.

**Initial Weight Values**

The initial weights are important. From eqs. (3.4.3.11) we see that $\Delta \omega_{ij} \propto g'(\cdot)$. The initial changes to the weights $\omega_{ij}$ should be large, i.e. $g'(\cdot)$ should be large. This means that the initial summed inputs (eq. 3.7) should be small. As a “rule-of-thumb” the size of the initial weights should be $\omega \approx \delta$/maximum “fan-in”), where $\delta \approx T/10$. The “temperature” $T$ sets the inverse slope for $g(\cdot)$ and the central part of $g'(\cdot)$ increases with $T$. A very low $T$ subsequently frustrates the weights. $T = 1$ is usually a good choice.

**Input Representation**

Input representation is crucial for achieving good performance [31]. Preprocessing of the data can be done in order to decrease the number of input units, thus decreasing the number of weights in the network. The representation of the data (using polar coordinates, etc.) influences the number of cuts needed and thus changes the number of hidden units. The network learns more easily, and less rescaling of parameters is needed, if all inputs are scaled to $\mathcal{O}(1)$. If one of the inputs is very much larger in magnitude than the others, the network will concentrate on that one and take a very long time to learn the others. This has to do with different “time constants” for the weights to reach their asymptotic values. Small inputs need a larger $\eta$ than large inputs, since the updatings (eq. (3.10)) are proportional to the input values. If the network needs to compare a very large input with a small one, the inputs to the hidden units should be of the same order for the two. This means that the weight from the large input should be small and the input from the small input should be large. Unfortunately eq. (3.10) promotes the opposite.

**Generalization**

Generalization is the networks performance on a set of test patterns $\mathcal{P}^{(p)}$ it has never seen before. The network learns from a number of training examples, creating a parameterization of the actual “function”. This parameterization is then used for the test patterns, much like interpolating in a lookup-table. The networks generalization performance is usually lower than its performance on the training set, although for very large
data sets the performances can be approximately equal. Typical learning curves are depicted in fig. 3.8. The lower curve (test set) reaches a maximum and then falls off, while the upper curve (training set) asymptotically climbs upwards. This is due to over learning, i.e. the network is learning the individual training examples, and subsequently losing generalization ability.

- **Optimal generalization**: The upper limit of generalization performance for a classification problem is given by the Bayes limit [33], which is the minimum overlap of the multidimensional distributions. This is illustrated in fig. 3.9 for the case of two Gaussian-like distributions in one dimension. For academic problems with overlapping Gaussian distributions, where Bayes limit is easily calculated, BP networks generalize to values very close to this limit [34]. However, for “real” problems with many dimensions, Bayes limit is usually practically impossible to calculate due to the CPU consumption involved.

- **Achieving optimal generalization**: The generalization performance depends mainly on the ratio $N_w/N_p$, where $N_w$ is the number of weights in the network and $N_p$ is the number of training patterns. For a feedforward network with one hidden layer of “thresholding” neurons the generalization error $\varepsilon$ is of order $O(N_w/N_p)$ [35]. As a “rule of thumb”, one should use at least 10 times more training patterns than the number of weights in the network.

- **Improving generalization**: Adding random noise to the input training patterns smears them and improves generalization. The improvement is however sensitive to the noise level. It is also possible to improve generalization performance in classification tasks after the initial training by using so-called border patterns. These are patterns in the training set whose outputs are close to the midpoint value of $g()$ and subsequently are close to the border between the classes. Additional training on such border patterns generally improves the classification performance.

![Figure 3.8: Typical behavior of learning curves.](image1)

![Figure 3.9: Two overlapping Gaussian distributions in one dimension. The optimal cut for the distributions is indicated.](image2)
Constraining the network

Restricting the number of weights in the network is important from many aspects. The generalization performance is better, without the need for a huge number of training patterns. The convergence time is shorter, since the number of weights to update is smaller. The latter is very important since one might need to try some different network architectures before the optimal performance is achieved. The best way to restrict the number of weights is to make use of knowledge about the task the network is supposed to perform.

- **Preprocessing:** Preprocessing the data can decrease the number of input units needed and simplify the problem such that fewer hidden units are needed.

- **Symmetries:** Consider the problem of discriminating between two patterns independent of translation and rotation. The possible patterns are represented in a $n \times m$ pixel matrix, with all possible rotations. These patterns can occur anywhere in a $N \times M$ input matrix ($N > n$, $M > m$), but the number of inputs needed to compute the task is only $n \cdot m$ (see fig. 3.10). In this case it is wise to use receptive fields [13, 36], i.e. overlapping fields with linked weights. The weights going from corresponding units in the fields to the hidden units are made equal, meaning that each updating to one of the weights is also made to the others. The effective number of weights is thus decreased from $N_w \sim O(N \times M \times N_h)$ down to $N_w \sim O(n \times m \times N_h)$, where $N_h$ is the number of hidden units in the “big” network and $N_h'$ is the number of hidden units that each receptive field connects to.

- **Limited input view:** Even though the problem does not have translational symmetry, it might be “local”, meaning that a hidden unit does not need to see all the input units. The hidden units are then connected to overlapping selective fields in the input layer.

- **Modularizing the network:** In several cases the actual task can be divided into subtasks, with known target values. For example, a vision task can be divided into a “what” and a “where” subtask [37]. Each subtask might need a small network and be easy to learn. The resulting networks are then merged, i.e. their output is used as input to a new network which is trained to perform the full task. Such modular networks often outperform networks that are trained on the full task from the start [15].

![Diagram](image.jpg)

Figure 3.10: Receptive fields; weights connecting corresponding parts of the receptive fields to hidden units are linked together. The two arrow patterns should give similar responses in the output units.
3.2.4 Example: Mirror Symmetry

To illustrate the power of including hidden units we can study the one-dimensional mirror symmetry problem [13], which deals with the symmetry of binary patterns \( \vec{x} = (x_1, x_2, ..., x_N) \), where \( x_i \in \{0, 1\} \). The task is to tell whether a specific pattern is symmetric with respect to reflection in its mid-point coordinate or not. For the case of 2 coordinates; the patterns (1,1), (0,0) are symmetric while (1,0) and (0,1) are not, which coincides with the XOR problem. This problem is not linearly separable [12] but can (in theory) be solved for any dimensionality of \( \vec{x} \) using only two hidden units. This is a consequence of the fact that all symmetric patterns lie in the same \([N/2]\)-dimensional multiplane \( \Pi_S \). To make a border around \( \Pi_S \), the network only has to use two hyperplanes, i.e. two hidden units, that are parallel to \( \Pi_S \) and slightly displaced from it (cf. fig. 3.4b).

A MLP easily learns the one-dimensional mirror symmetry for moderate input dimensions. A possible solution for 4 inputs is shown in fig. 3.11. The weight vectors \( \vec{\omega}_1 = (-1, 2, -4, 4, 2) \) and \( \vec{\omega}_2 = (1, -2, -4, 4, 2) \) for the hidden units are linear combinations of \( \vec{x}_1 = (1, 1, -1, -1) \) and \( \vec{x}_2 = (1, -1, 1, -1) \), which span the subspace orthogonal to \( \Pi_S \) (the coordinate \( \omega_5 \) is the threshold). The output unit is performing a simple OR type function on the two hidden units, it fires only when \( h_2 \) is active and \( h_1 \) is inactive.

If we train a MLP with 10 hidden units and apply the pruning procedure above (cf. eq. (3.22)), the net decreases the number of hidden units to 2 by cutting the weights to the other units, showing that only 2 hidden units are needed.

![Diagram](image)

Figure 3.11: A three-layered MLP that solves the one-dimensional mirror symmetry problem with 4 input nodes.

4 Self-organization

The multilayer perceptron is very appealing with its simple structure and proven impressive track record. However, it suffers from some limitations:

- The learning has to be done in a supervised manner, the features \( o_i \) have to be known beforehand. This is the most severe limitation.
- It is somewhat cumbersome to analyze the representation that the hidden nodes have built up. For sigmoidal units, the decision regions are not strict polyhedral regions, but rather "polyhedral"s with smooth corners.

A self-organizing (SO) network operates in a different way which complements the multilayer perceptron. Self-organizing networks are networks that organize themselves according to the "natural structure" of the data without supervision. This "structure" can be clusters, principal components, prototypes, and/or other typical features. A SO network has usually only one layer of units besides the input units and we will term these units feature units from now on and denote them by \( h_j \) (see fig. 4.1). The resulting
output from these feature units is a reduced representation of the data, reflecting the main features of the data — SO networks can thus be used as feature extractors on data.

Self-organizing networks exist on two levels of sophistication. The simple one is the competitive or winner-takes-all, where only one feature unit is allowed to be on at the time, whereas the more sophisticated collective version allows several units to react at the same time and utilizes this to extract relationships in the data. There exist a number of SO algorithms but we will only discuss a few of them here (see ref. [1] for a general review).

Figure 4.1: A self-organizing network. Note that possible interactions among the feature nodes are of feedback nature.

### 4.1 Competitive Self-organization

In a competitive network a pattern \( \mathbf{z}^{(p)} \) is presented and the responses of the feature nodes \( h_j \) are computed,

\[
h_j = G(\mathbf{z}^{(p)})
\]

where the response function \( G() \) can be of different form, depending on how the input is coded. For each presented pattern, a winner unit \( h_m \) is picked out among the feature nodes.

\[
h_m = \max_j(h_j)
\]

The weights of this winner unit are then updated such that \( h_m \) becomes even more representative of the presented pattern \( \mathbf{z}^{(p)} \). How this updating is done depends on how the "representativity" is measured (clusters, principal components, etc.).

#### 4.1.1 Vector Quantization

Perhaps the most used form of competitive self-organization is vector quantization (VQ) [38], which uses an algorithm identical to the adaptive k-means clustering algorithm [39]. In vector quantization, each unit \( h_j \) is associated with a weight vector \( \mathbf{w}_j = (\omega_{j1}, \omega_{j2}, \ldots, \omega_{jN}) \) of the same dimensionality as the input patterns \( \mathbf{z}^{(p)} = (z^{(p)}_1, z^{(p)}_2, \ldots, z^{(p)}_N) \). The "representativity" is measured in (Euclidean) distance between \( \mathbf{w}_j \) and \( \mathbf{z}^{(p)} \). For each presented pattern \( \mathbf{z}^{(p)} \), the distance between the weights vectors and the pattern is computed and the winner unit is the unit closest to the pattern.

\[
h_m = \min_j(|\mathbf{w}_j - \mathbf{z}|)
\]

This is analogous to using a response function

\[
G(\mathbf{z}) = \exp(-|\mathbf{w}_j - \mathbf{z}|^2 / T)
\]
for the feature units, where the "temperature" $T$ sets the width of the Gaussian. The weight vector $\tilde{w}_m$ belonging to the winner unit is then moved closer to $\tilde{x}^{(p)}$ according to

$$\Delta \tilde{w}_m = \eta (\tilde{x}^{(p)} - \tilde{w}_m)$$  \hspace{2cm} (4.5)$$

where $\eta$ is the learning rate. This basic algorithm is summarized in fig. 4.3.

More insight into the VQ learning algorithm can be gained by studying a more general formulation of eq. (4.5)

$$\Delta \tilde{w}_j = \eta \delta_{jm} (\tilde{x}^{(p)} - \tilde{w}_j)$$  \hspace{2cm} (4.6)$$

where $\delta_{jm}$ is Kronecker's delta. This corresponds to gradient descent on an error function defined as

$$E^{(p)} = \frac{1}{2} \sum_j \delta_{jm} (\tilde{x}^{(p)} - \tilde{w}_j)^2$$  \hspace{2cm} (4.7)$$

Summing (4.7) over all patterns $p$ we get

$$E = \sum_p E^{(p)} = \frac{1}{2} \sum_{p,j} \delta_{jm} (\tilde{x}^{(p)} - \tilde{w}_j)^2$$  \hspace{2cm} (4.8)$$

which is minimized when $\tilde{w}_j = \langle \tilde{x}^{(p)} \rangle$ for all $p \in \mathcal{J}$, where $\mathcal{J}$ is the set of patterns giving unit $h_j$ as a winner. The weight vectors will thus converge towards "cluster centers" in the data distributions. The $\tilde{w}_j$'s quantize (hence the name) the input space into polyhedral "compartments" centered around $\tilde{w}_j$ (see fig. 4.2c). The final network will be extremely easy to analyze; one simply inspects the weight vectors $\tilde{w}_j$ and they will mimic the specific features that the units are most sensitive to. This is the key virtue of this method.

Figure 4.2: (a) A weight vector $\tilde{w}_j$ lining up with an input vector $\tilde{x}$. (b) Three feature vectors averaging different clusters of input categories. (c) The resulting quantization of the input space; each dot corresponds to one weight vector $\tilde{w}_j$.

1. Initialize $\tilde{w}_j$'s.
2. Repeat until $\tilde{w}_j$'s have converged:
   2.1 Pick pattern $p$ from the data set.
   2.2 Present input $\tilde{x}^{(p)}$ and calculate $h_j$
      according to eqs. (4.1,4.4).
   2.3 Pick out the winner unit $h_m$ (eq. (4.2)).
   2.4 Update the weight vector of the winner according to
      $$\tilde{w}_m(t + 1) = \tilde{w}_m(t) + \eta (\tilde{x}^{(p)} - \tilde{w}_m).$$

Figure 4.3: Vector Quantization.
4.1.2 Learning Vector Quantization

Although vector quantization is an unsupervised learning algorithm, for classification purpose one can augment it with supervised learning for fine tuning the units specific to certain features [38]. This is called learning vector quantization (LVQ) and amounts to learning correct answers and unlearning incorrect answers. The only change in the learning algorithm (fig. 4.3) is that the updating equation (4.5) is changed into

\[
\Delta \omega_m = \begin{cases} 
+ \eta (\tilde{z}^{(p)} - \omega_m) & \text{if } \tilde{z}^{(p)} \text{ is correctly classified} \\
- \eta (\tilde{z}^{(p)} - \omega_m) & \text{otherwise}
\end{cases}
\] (4.9)

The relative performance of LVQ and BP on pattern classification problems strongly depends upon the dimensionality of the problem. LVQ is more efficient and faster for low dimensional problems whereas BP is more economical for high dimensional problems. This can be seen from the following argument. Consider the case of overlapping distributions in some \( N \)-dimensional space and let us assume that these are complex in the sense that the optimal "cut" between them is not one simple hyperplane. LVQ quantizes the input space by using "Gaussians" (cf. eq. (4.4)) and fig. 4.2c) and thus has to fill a volume in the input space to produce a good "cut". The BP feed-forward network on the other hand divides the input space using \( (N-1) \)-dimensional hyperplanes (see fig. 3.4). The number of units needed for LVQ to perform well on a specific task should therefore scale like \( a^N \), while the number of units needed in the BP network should scale like \( a^{(N-1)} \). From "benchmark" studies in refs. [34, 40] it is clear that LVQ has difficulties in reaching Bayes limit for synthetic problems consisting of overlapping high dimensional Gaussians distributions in contrast to BP and other supervised learning algorithms based on sigmoidal units like the Boltzmann machine [41] and Mean Field Theory learning [34, 42]6).

In high energy physics ANN applications where comparisons have been made between LVQ and BP [43, 44], LVQ has never been found to perform better than BP.

4.1.3 Practical Hints

Initial Weight Values

- \( VQ \): The weights are initialized with random values, as in the BP case.
- \( LVQ \): The simplest way to initialize the weights is to give them the values of some randomly picked data points.

Learning Parameters

The parameters are \( \eta \) and \( T \) (eq. (4.4)). The temperature \( T \) has only practical importance in computer simulations. It determines the "window" of each unit. It should be chosen \( T \sim \mathcal{O}(\tilde{z}^2) \) such that the activations \( G(\tilde{z}) \sim \mathcal{O}(10^{-1}) \).

- \( VQ \): The learning rate \( \eta \) should be large in the beginning such that large changes \( \Delta \omega_j \) can be achieved. It is then lowered as the network converges in order for the network to settle into a stable state.
- \( LVQ \): If the weights have been initialized as described above, \( \eta \) must be small all the time, since the initial weights are already quite close to the final solution. Also in this case \( \eta \) is allowed to decrease.

4.2 Collective Self-organization

The previous two sections dealt with competitive self-organizing, where only one unit was selected the winner. In the following section we discuss the case where several units are allowed to react and adapt to a given input \( \tilde{z}^{(p)} \). The general algorithm introduces a neighborhood function \( \Lambda(h, j) \) defining the group of units that are allowed to be updated. The neighborhood function can depend both on the activation \( h \) and the index \( j \) for the

---

6) The conclusions with respect to performance of BP in [40] are in error since the architecture used for BP was inappropriate for the problem. We refer to ref. [34] for a proper discussion of this issue.
unit. In the competitive case the neighborhood function is simply the Kronecker delta $\delta_{jm}$ (cf. eq. (4.6)). This collective updating is then used to extract information on internal relations in the input data.

In section 6.2 (and 8.2.2) an alternative collective self-organizing algorithm is presented where the weights are updated in proportion to the relative strengths of the feature nodes $h_j$ (cf. the Potts updating of eq. (3.14)).

4.2.1 Feature Maps

Feature mapping is a collective version of vector quantization that makes topologically correct maps of the input space onto a "feature map". For this, a geometrical topology has to be defined for the feature units $h_j$, usually a $N$-dimensional lattice (see fig.4.1 where $N = 2$). The error function (4.7) is modified into [45]

$$E^{(p)} = \frac{1}{2} \sum_j \Lambda_{jm}(\tilde{x}^{(p)} - \tilde{w}_j)^2$$

(4.10)

where $\Lambda_{jm}$ is a function of the (Euclidean) distance $d_{jm}$ between units $j$ and $m$ in the lattice. The typical shape for $\Lambda_{jm}$ is Gaussian

$$\Lambda_{jm} \propto e^{-d_{jm}^2/\lambda^2}$$

(4.11)

where $\lambda$ is a parameter determining the width of the neighborhood.

When a pattern $\tilde{x}^{(p)}$ is presented to the network, the winner unit $h_m$ is selected and the weight vectors are updated according to

$$\Delta \tilde{w}_j = \eta \Lambda_{jm}(\tilde{x}^{(p)} - \tilde{w}_j)$$

(4.12)

where the neighborhood width $\lambda$ (eq. (4.11)) is decreased as training proceeds. This causes close neighbors to the winner unit to be updated in the same direction as the winner. In this way neighboring units will end up with similar weight vectors and the resulting network will be topologically correct (if possible), i.e. neighboring points in the input space activate neighboring units in the network.

1. Initialize the $\tilde{w}_j$'s.
2. Repeat until $\tilde{w}_j$'s have converged:
   2.1 Pick pattern $p$ from the data set.
   2.2 Present input $\tilde{x}^{(p)}$ and calculate $h_j$ according to eqs. (4.1,4.4).
   2.3 Pick out the winner unit $h_m$ (eq. (4.2)).
   2.4 Update the weights according to
      $$\tilde{w}_j(t + 1) = \tilde{w}_j(t) + \eta \Lambda_{jm}(\tilde{x}^{(p)} - \tilde{w}_j)$$

where $\Lambda_{jm}$ is given by eq. (4.11).

Figure 4.4: Feature mapping.

4.2.2 Practical Issues and "Rules of Thumb"

Even though the general mapping algorithm (fig. 4.4) is very simple, there exist a number of factors that are crucial for the final result. These include the learning parameters and the neighborhood function $\Lambda_{jm}$.

Convergence of the Map

An important theoretical question is if the algorithm converges to a stable state at all. There is no proof for the general $N$-dimensional map, but it can be shown for one- and two-dimensional maps that they will converge to an equilibrium state [45] if the learning rate goes to zero with time ($\eta(t) \rightarrow 0$).
Features of the Map

The quality of the resulting map is controlled by the topology of the map in relation to the topology of the problem (see examples below). The dimensionality and shape of the feature unit setup must resemble the dimensionality and shape of the problem to produce a high quality mapping.

- **Dimensionality of the map:** For the mapping to be useful, the feature map must be of lower dimensionality than the input space. The dimension of the problem is however not necessarily equal to the input dimension, the data points could be lying on a \((N - 1)\) dimensional hypersurface and the problem dimensionality would thus be \((N - 1)\). To map all the features of the problem, the feature map has to have the same dimensionality as the problem. Usually, one needs to map only the most important features of the data and a low-dimensional map is sufficient.

- **Shape of the map:** The shape of the map should be as close to the shape of the data distribution as possible (see fig. 4.5). A triangular data distribution maps optimally on a triangular feature map. Furthermore, for closed surfaces, the map must have periodical boundaries for a topologically correct mapping to be possible. In most cases, however, the topology of the data is unknown and the aim of the network is to extract this knowledge. In such cases one just has to make a “qualified” guess, which can be verified/falsified and followed by a new mapping.

- **The density of units:** The resulting density \(P_\omega(\bar{z})\) of weight vectors will reflect the density \(P_p(\bar{z})\) of data points (see fig. 4.6). The feature unit density \(P_\omega\) is usually an increasing function of \(P_p\), but the relationship is not linear. For a one-dimensional map the relationship can be shown to be \(P_\omega \propto P_p^{2/3}\) [46].

The Neighborhood

The shape and width of \(\Lambda_{jm}\) is important for good performance. The shape influences the handling of “conflicts” and the width controls the plasticity of the map.

- **Shape:** \(\Lambda_{jm}\) is usually either a Gaussian (eq. (4.11)) or a simple square function

\[
\Lambda(j, m) = \begin{cases} 
1 & \text{if } d_{jm} \leq \lambda \\
0 & \text{otherwise}
\end{cases} \quad (4.13)
\]

where \(\lambda\) is the width\(^7\). For the one-dimensional map \(\Lambda_{jm}\) should be a convex function with maximum for \(j = m\) in order to speed up convergence. If \(\Lambda_{jm}\) is concave, a number of metastable states occur and convergence is very slow [47]. Also, if \(\Lambda_{jm}\) is asymmetric, “conflicts” are removed faster and convergence speeded up [48].

- **Width:** The width \(\lambda\) of the neighborhood determines the plasticity of the net. If \(\lambda\) is large, the feature units are strongly coupled together, the network is “stiff”. If \(\lambda\) is zero, the feature units are completely free and are allowed to converge to their respective cluster centers, the net is “soft”. In the beginning of learning, \(\lambda\) should be large (\(O(\text{size of map})\)) in order for the network to find the general location of the data. As learning proceeds, \(\lambda\) is decreased (linearly) such that the units can map more details of the data distribution. In cases where the data distributions have a larger dimensionality than the feature map, the final quality of the map can be controlled with \(\lambda\). This is illustrated in figure 4.5 that shows a map where \(\lambda \rightarrow 0\) as training proceeds, the feature units converge fully and the map shows much detail, but the topological order is not correct.

Initial Weight Values

The weight values should be initialized such that \(\tilde{\omega}_j \approx \langle \bar{z} \rangle\).

\(^7\) Using the square form is referred to as the “short cut” algorithm.
Learning Parameters

The principal parameters for the feature map are $\lambda$, which is discussed above, $\eta$ and the "temperature" $T$ (eq. (4.4)). The same comments on $T$ as for VQ and LVQ apply also here. The initial $\eta$ should be large ($0.5 < \eta < 1.0$) such that large changes can be made to the weights. It is then lowered during training such that a stable equilibrium is achieved (see above on convergence).

4.2.3 Examples: Shapes and Densities of Distributions

Triangular Distribution

The input to the network consists of points $(x_1, x_2)$ that are evenly distributed over a triangular area (fig. 4.5a). The feature map is one-dimensional with 50 units. The weights are initialized with random values. The formation of the map is shown at five different stages. It is obvious that a one-dimensional network cannot capture the topology of a two-dimensional distribution perfectly, but the network tries to fill the triangle as well as possible. If the neighborhood size $\lambda$ is allowed to shrink to zero (fig. 4.5f) the network units are allowed to converge fully, resulting in the loss of topological order. However, if $\lambda$ is small but non-zero (fig. 4.5d), the main topology can be extracted, but with a loss of detail. Detail is thus achieved through loss of topological information and vice versa.

![Triangular Distribution Examples](image)

Figure 4.5: Mapping a two-dimensional triangular distribution onto a one-dimensional network with 50 units; from (a) random initial weights to (f) final state. The neighborhood size $\lambda$ decreases linearly from 15 down to 0.

Gaussian Distribution

The input to the network are points $(x_1, x_2)$ drawn from a two-dimensional Gaussian distribution (fig. 4.6a). The topology of the network is now a $10 \times 10$ lattice. The weights are initialized with random values around the center point of the distribution. As the map is being formed several "conflicts" can occur and have to be removed. It is often the removal of these conflicts that takes most time. Convergence can thus be considerably speeded up by smart choices (and adaption during learning) of parameters so that conflicts are more easily solved. The final density of feature units increases towards the center point — the average distance between units reflects the density of data points.
Figure 4.6: The formation of a map of a two-dimensional Gaussian distribution; (a) distribution and initial random map, (f) final map. The neighborhood size $\lambda$ decreases linearly from 5 to 0.

5 Feed-back Networks

Feed-back networks appear in the context of associative memories (the Hopfield model [49, 50]) and difficult optimization problems [51, 52] but also in feature recognition applications (the Boltzmann machine [41] and its mean field approximation [34]). Simple models for magnetic systems have a lot in common with feed-back networks and have hence been the source of much inspiration. We therefore start this section by familiarizing the reader with Ising models for magnetic systems.

5.1 Magnetic Systems

The Ising model describes a magnetic system in terms of binary spins $s_i \in \{-1, 1\}$ that are effective variables for the individual atoms. The two spin states at each site represent the possible magnetization directions. The Ising system is governed by the energy function

$$E = -\frac{J}{2} \sum_i s_i s_{i+1}$$

(5.1)

where nearest neighbors interact pairwise with a constant attractive force of strength $J$. The lowest energy state is reached by iterative updating of

$$s_i = \text{sgn}[J(s_{i-1} + s_{i+1})]$$

(5.2)

which leads to a state where all spins point in one of the two possible directions (see fig. 5.1a). If the system is embedded in a temperature environment (the situation in fig. 5.1a assumes $T=0$) fluctuations will appear subject to the Boltzmann distribution

$$P(\vec{s}) \propto e^{-E(\vec{s})/T}$$

(5.3)

where the dynamics of eq. (5.2) is replaced by some stochastic procedure leading to fluctuating configurations (see fig. 5.1b).

8) There is also a feed-back version of back-propagation: recurrent back-propagation [53].
all \( s_i = +1 \) \( \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \) \( \langle s_i \rangle > 0 \) 

all \( s_i = -1 \) \( \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \) \( \langle s_i \rangle < 0 \) 

(a) \hspace{5cm} (b)

Figure 5.1: (a) The two possible E=0 states for the Ising model at temperature T=0. (b) Two \( T \neq 0 \) configurations.

Depending on \( T \), the time-averaged values \( \langle s_i \rangle \) can point in one direction or the other. At very high temperatures, above the phase transition point, there is no alignment at all. All the spins are completely random. A phase transition behavior from an disordered phase to an ordered one is depicted in fig. 5.2.

\[ E \]

\[ \text{disorder} \rightarrow \text{order} \]

\[ T \]

\[ \text{critical temperature } T_c \]

Figure 5.2: \( E \) as a function of \( T \) illustrating a phase transition.

Such transitional phenomena represent global properties of the system. Following the dynamics of individual spins does not tell us very much. The transition into the ordered phase will play an important role in feed-back network applications below — information is order.

Similar features hold for more realistic modeling of magnets in three dimensions. It is interesting and reassuring that spin models are effective theories; all atomic physics details are lumped into effective spin variables and the energy function \( E \). This is sufficient for capturing the global properties, or collective properties, of the system.

Let us generalize the Ising model to a spin glass system\(^9\) by:

- Allowing for non-local interactions \( s_i s_{i+1} \rightarrow s_i s_j \) for all \( j \neq i \).
- Allowing for different, but symmetric, bond strengths \( J \rightarrow \omega_{ij} = \omega_{ji} \) between \( s_i \) and \( s_j \).

Thus eq. (5.1) is replaced by

\[ E = -\frac{1}{2} \sum_{i \neq j} \sum_j \omega_{ij} s_i s_j \]

(5.4)

The fact that the bonds can be of different signs has the effect that such systems contain conflicting interests — frustration. This is illustrated in fig. 5.3; all the constraints (spins connected with positive bonds pointing in the same direction) cannot be satisfied simultaneously. This leads to many “ground states” with almost the same energy in the ordered phase \( (T < T_c) \). There exist \( \approx e^{0.2N} \) such “almost ground states” for a \( N \) spin

\(^9\) Spin-glasses model certain alloys like AuFe.
system. It is suggestive that such a system could be exploited in information technology — a feedback network.

Figure 5.3: 4 spins connected with different sign bonds.

5.2 The Hopfield Model

The Hopfield model [49, 50] is based on the energy function of eq. (5.4) with binary neurons \( s_i = \pm 1 \). By appropriate choice of \( \omega_{ij} \) the idea is to let the model function as an associative memory (cf. Section 1.2.2). The dynamics that locally minimizes eq. (5.4) is given by (cf. eq. (5.2)).

\[
    s_i = \text{sgn} \left[ \sum_{j \neq i} \omega_{ij} s_j \right] 
\]

(5.5)

Given a set of \( N_p \) patterns \( \tilde{x}^{(p)} \) (\( \tilde{x}^{(p)} = (x_1^{(p)}, x_2^{(p)}, ..., x_N^{(p)}); p = 1, ..., N_p \)), the Hebb rule [10]

\[
    \omega_{ij} = \sum_{p=1}^{N_p} x_i^{(p)} x_j^{(p)} 
\]

(5.6)

is used for learning. For example with

\[
    \tilde{x}^{(1)} = (1, 1, -1, 1, -1) \\
    \tilde{x}^{(2)} = (-1, 1, -1, -1, -1)
\]

one gets

\[
    \omega_{12} = 1 \cdot 1 + (-1) \cdot 1 = 0 \\
    \omega_{23} = 1 \cdot (-1) + 1 \cdot (-1) = -2
\]

etc.. Thus the Hebb rule increases weights (synapses) between neurons whose activities are correlated and decreases weights between anticorrelated neurons.

With \( \{0,1\} \)-neurons, the Hebb rule takes the form

\[
    \omega_{ij} = \sum_{p=1}^{N_p} (x_i^{(p)} - 1)(x_j^{(p)} - 1) 
\]

(5.7)

The Hebb rule is very appealing since it is both local and incremental. Since \( \omega_{ij} \) can take either sign it gives rise to an "interesting" energy landscape like in the spin glass case. To understand how the \( \tilde{x}^{(p)} \)'s are related to the stable states of the energy of eq. (5.1), we compute the local field \( a_i = \tilde{w}_i \cdot \tilde{s} = \sum_j \omega_{ij} s_j \) when \( \tilde{s} = \tilde{x}^{(q)} \) (\( q = \) one of the stored states) using the Hebb rule for \( \omega_{ij} \)

\[
    a_i = \sum_{j=1}^{N} \omega_{ij} s_j = \sum_{j=1}^{N_p} \sum_{p=1}^{N} x_i^{(p)} x_j^{(p)}^{(q)} x_j^{(q)}
\]
\[= [p = q] + [p \neq q] \]
\[= \bar{x}_i^{(q)} \sum_{j=1}^{N} \bar{x}_j^{(q)} + \sum_{p \neq q} \bar{x}_i^{(p)} \sum_{j=1}^{N} \bar{x}_j^{(p)} \bar{x}_j^{(q)} \]
\[= N\bar{x}_i^{(q)} + \delta_i \]

(5.8)

where we have used \(|\bar{x}^{(q)}|^2 = N\). The two terms in eq. (5.8) represent signal and noise ("crosstalk") with respect to having \(\bar{s} = \bar{x}^{(q)}\) as a stable state to eq. (5.5). For completely random patterns \((x_i = \text{rand} \{-1,1\})\) one has \(\langle \delta_i \rangle = 0\) and \(\langle \delta_i^2 \rangle = N(N_p - 1)\), which in the \(N \to \infty\) limit yields a signal/noise ratio

\[R = \frac{\text{signal}}{\text{noise}} = \frac{N}{\sqrt{N(N_p - 1)}} = \sqrt{\frac{N}{N_p}} \]

(5.9)

\(R\) sets the capacity limit for storing patterns in the Hopfield network. For random patterns the capacity is \(N_p/N \approx 0.14\) [54], corresponding to \(R = 2.7\). Attempting to store more patterns than \(0.14N\) will cause "spurious states" (non-memory stable states) \(\bar{x}^{(p)}\) to appear.

The key advantages with a "Hopfield memory" is that it is associative and robust. If a pattern \(\bar{x}^{(p)}\) is presented to the network where some bits are distorted as compared to a stored pattern \(\bar{x}^{(q)}\), the dynamics of eq. (5.2) will complete the pattern (or "correct" it). Also, if some of the weights \(\omega_{ij}\) are lost, the network will still perform well, since the memory is distributed over all the weights. It is illuminating to think about this in terms of dynamics in the energy landscape\(^{10}\). In fig. 5.4 a schematic one-dimensional view of \(E\) as a function of different configurations \(\bar{s}\) is shown. Stored patterns are represented by local minima in \(E\). A distorted pattern (e.g. some of the bits off as compared to the memorized one) is a state located uphill from the pure pattern. The updating dynamics of eq. (5.5) makes the network slide downhill towards the correct pattern.

Figure 5.4: Schematic energy landscape with patterns stored with Hebb rule indicated. Open and closed circles denote starting and final states respectively.

The basic steps for the Hopfield model are in fig. 5.5.

Various modifications of eq. (5.6) have been suggested to improve the storage capacity. They fall into two classes: local [55, 56] and non-local [57]. In non-local approaches one computes a pattern correlation matrix \(C_{pq}\)

\[C_{pq} = \sum_i \bar{x}_i^{(p)} \bar{x}_i^{(q)} \]

(5.10)

\(^{10}\) Not the same energy landscape as for BP networks, where \(E\) as a function of the \(\omega_{ij}\)'s is considered.
1. "Program" $\omega_{ij}$ using the Hebb rule (eq. (5.6)).

2. Choose a starting state $\mathbf{s} \approx \mathbf{z}^{(p)}$.

3. Repeat until $\Delta \mathbf{s} = 0$:
   3.1 Pick a neuron $s_i$ at random
   3.2 Update according to
   $$s_i = \text{sgn}[\sum_{j \neq i} \omega_{ij} s_j]$$

Figure 5.5: A Hopfield model algorithm

which is then used in a modified Hebb rule

$$\omega_{ij} = \sum_{pn} C^{-1} z_{i}^{(p)} z_{j}^{(n)}$$

Clearly this procedure results in a more optimized energy landscape. However this is at the expense of not having a local learning rule, which is attractive both from the point of view of biological plausibility and feasibility of designing custom made VLSI hardware.

It is demonstrated in ref. [55] that the performance improves if the rule of eq. (5.6) is supplemented by "unlearning" of the spurious states $\xi^{(p)}$. It is also possible to let the network run freely in order to detect the spurious states and remove them "by hand";

$$\omega_{ij} = -\epsilon \sum_{p=1}^{N^*_p} \xi^{(p)}_{i} \xi^{(p)}_{j}$$

where $\epsilon$ is a parameter and $N^*_p$ the number of spurious states. Subsequent work has verified the power of this method with storage capacities in the range $30 - 40\%$ as a result [58].

It has been suggested that during REM sleep, mammals let the brain run freely (dream) as a means for removing spurious unwanted memories [59]. This "unlearning" procedure is closely related to the Boltzmann machine learning.

5.3 The Boltzmann Machine

The Hopfield model is very appealing with its local learning rule. However, as stated above, it suffers from two shortcomings:
- Learning is not optimized — spurious states are not removed.
- It cannot be extended to difficult recognition tasks since Hebbian learning cannot handle hidden units — $h^{(p)}$ unknown in $\sum_s h^{(p)}$.

The Boltzmann machine (BZ) [41] is a learning algorithm that handles these problem while still preserving local learning. It was originally intended for pattern recognition applications but can also be "extended" to the case of no hidden units in the context of content addressable memories [62]. BZ learning learning is unacceptably slow in serial simulations, but (as will be discussed later) its deterministic version, mean field theory (MFT) learning, is quite competitive. Furthermore, it reduces to the back-propagation algorithm in the limit of few outputs. The dynamics of BZ is based on the Hopfield energy function (eq.(5.4)). The model learns by making an internal representation of its environment. The learning procedure changes weights so as to minimize the distance between two probability distributions, as measured by the $G$-function or the so-called Kullback measure [18] (cf. eq. (3.13))

$$G = \sum_{\alpha} P_{\alpha} \log \frac{P_{\alpha}}{P'_{\alpha}}$$ (5.13)
where \( P_\alpha \) is the probability that the visible units, input and output (see fig. 5.6) are collectively in state \( \alpha \) when their states are determined by the environment. \( P_\alpha \) represents the desired probabilities for these states. The corresponding probabilities when the network runs freely are denoted \( P'_\alpha \). \( G \) is zero if and only if the distributions are identical; otherwise it is positive. The word "free" either means that all visible units are free or that only the output units are free. In our presentation we will stick to the latter alternative, the formalism is basically the same.

![Visible and hidden (internal) units in a Boltzmann machine.](image)

The probability \( P'_\alpha \) is given in terms of a Boltzmann distribution of the energy \( E_{\alpha \beta} \) as

\[
P'_\alpha = \frac{1}{Z} \sum_\beta e^{-E_{\alpha \beta}/T} \tag{5.14}
\]

where \( \alpha \) denotes states of the visible units and \( \beta \) of hidden ones.

\( Z \) is the partition function over all possible states of the network

\[
Z = \sum_{\alpha \beta} e^{-E_{\alpha \beta}/T} \tag{5.15}
\]

\( G \) can be rewritten as

\[
G = \sum_\alpha P_\alpha \log \frac{P_\alpha}{P'_\alpha} = \sum_\alpha P_\alpha (\log P_\alpha + \log Z - \log Z_\alpha) \tag{5.16}
\]

where \( Z_\alpha \) is the partition function summed over all possible states when the visible units are clamped in state \( \alpha \);

\[
Z_\alpha = \sum_{\beta} e^{-E_{\alpha \beta}/T} \tag{5.17}
\]

In order to minimize \( G \) we need \( \partial G / \partial \omega_{ij} \),

\[
\frac{\partial G}{\partial \omega_{ij}} = \sum_\alpha P_\alpha \left( \frac{\partial \log Z}{\partial \omega_{ij}} - \frac{\partial \log Z_\alpha}{\partial \omega_{ij}} \right) \tag{5.18}
\]

where \( P_\alpha \) is constant. For the two terms one gets

\[
\frac{\partial \log Z}{\partial \omega_{ij}} = \frac{1}{Z} \sum_{\alpha \beta} s_i s_j e^{E_{\alpha \beta}/T} \left( -\frac{1}{T} \right) = -\frac{\langle s_i s_j \rangle}{T} \tag{5.19}
\]
and
\[
\frac{\partial \log Z_\alpha}{\partial \omega_{ij}} = \frac{1}{Z} \sum_\beta s_i s_j e^{E_{\alpha\beta}/T} \cdot \left( -\frac{1}{T} \right) = -\frac{\langle s_i s_j \rangle_\alpha}{T}
\]
(5.20)

and thus for \( \partial G/\partial \omega_{ij} \) one has
\[
\frac{\partial G}{\partial \omega_{ij}} = -\frac{1}{T} \sum_\alpha P_\alpha [\langle s_i s_j \rangle - \langle s_i s_j \rangle_\alpha]
\]
(5.21)

Usually one wants all the \( M \) stored patterns to have the same probability \( P_\alpha = 1/M \), leading to
\[
\frac{\partial G}{\partial \omega_{ij}} \propto -\langle (s_i s_j)^c \rangle - \langle s_i s_j \rangle
\]
(5.22)

where we have introduced the notation \( \langle s_i s_j \rangle^c = \langle s_i s_j \rangle_\alpha \) (\( c = \) "clamped"). Gradient descent on \( G \) gives the updating rule
\[
\Delta \omega_{ij} = \eta \langle (s_i s_j)^c \rangle - \langle s_i s_j \rangle
\]
(5.23)

where \( \eta \) is the learning rate. At some low temperature \( T_o \), \( \langle s_i s_j \rangle^c \) is computed when both input and output nodes are clamped to an input pattern and \( \langle s_i s_j \rangle \) is computed when only the input nodes are clamped. The first term in eq. (5.23) corresponds to Hebbian learning since the visible units are clamped to the patterns to be learned. The second term is closely related to the REM sleep "unlearning" in eq. (5.12) — spurious states are removed. In this way the Boltzmann machine optimizes learning at the same time as it handles hidden units. The calculation of \( \langle s_i s_j \rangle^c \) and \( \langle s_i s_j \rangle \) should take place at an approximate minimum of \( E \) (eq. (5.4)). In order to ensure this one needs to anneal the system by thermalizing configurations with some stochastic method like the heat bath algorithm [27] for a sequence of decreasing temperatures
\[
P(s_i \rightarrow 1) = [1 + \exp(\sum_j \omega_{ij} s_j/T)]^{-1}
\]
(5.24)

1. Initialize \( \omega_{ij} \) with \( \pm \) random values.
2. Repeat until \( \omega_{ij} \)'s have converged:
   2.1 Clamped phase. The values of the input and output units of the network are clamped to a training pattern, and for a sequence of decreasing temperatures \( T_n > T_{n-1} > \ldots > T_o \), the network of eq. (3.1) is allowed to relax. At \( T = T_o \) statistics are collected for the correlations \( \langle s_i s_j \rangle^c \).
   2.2 Free phase. The same procedure as above, but this time the network runs freely or with only the input units clamped. Correlations \( \langle s_i s_j \rangle \) are measured at \( T = T_o \).
   2.3 Update \( \omega_{ij} \) according to
   \[
   \omega_{ij}(t + 1) = \omega_{ij}(t) + \eta (\langle s_i s_j \rangle^c - \langle s_i s_j \rangle)
   \]

Figure 5.7: A Boltzmann machine algorithm.

The Boltzmann machine algorithm is summarized in fig. 5.7. Needless to say this is a very CPU time consuming process due to the annealing and thermalization. In the next section we will discuss the mean field approximation, which replaces the stochastic updating of eq. (5.24) with a set of deterministic equations.
5.4 The Mean Field Approximation

Consider the Hopfield energy function (eq. (5.4)) and the corresponding updating equation (eq. (5.5)), which in terms of the local field \( u_i \) (cf. eq. (3.7))

\[
\frac{\partial E}{\partial s_i} = \sum_{j \neq i} \omega_{ij} s_j
\]

reads

\[
s_i = \text{sgn}(u_i)
\]

This updating equation is based on gradient descent and is hence suitable for associative memory applications; given an initial configuration it takes us to the closest local minimum. Other applications like feature recognition in BZ or optimization problems (see next section) require that an approximate global minimum of eq. (5.4) is reached. In BZ we use the very time consuming simulated annealing stochastic procedure. The key idea in the MFT approach is to approximate \( u_i \) by its thermal average

\[
u_i \approx \langle u_i \rangle_T = \sum_{j \neq i} \omega_{ij} \langle s_j \rangle_T = \sum_{j \neq i} \omega_{ij} v_j
\]

where \( v_j = \langle s_j \rangle_T \). We will not derive the MFT approximation here in detail but refer the reader to refs. [42, 52, 60]. To obtain this approximation one starts by rewriting the partition function sum \( Z \) (eq. (5.15)) as

\[
Z = \text{const} \cdot \int (\prod_i^N dv_i) e^{-\frac{1}{2} \sum_{ij} \omega_{ij} v_i v_j + \sum_i \log \cosh(\sum_j \omega_{ij} v_j / T)}
\]

One then assumes (saddle point approximation) that the contribution to \( Z \) is dominated by the minima of the exponent \( E' \) of eq. (5.28). Taking \( \partial E' / \partial v_i = 0 \) yields the MFT equations

\[
v_i = \tanh(\sum_j \omega_{ij} v_j / T)
\]

In other words; the stochastic updating process is being emulated by a set of deterministic equations with sigmoid gain functions (cf. eq. (2.2.2.3)). At \( T=0 \) the Hopfield step-function updating rule is recovered. In general, this approximation is supposed to hold for large \( N \). In ANN applications the approximation is extremely reliable (see e.g. refs. [42, 52]). For \([0,1]\) neurons the MFT equations read

\[
v_i = \frac{1}{2} [1 + \tanh(\sum_j \omega_{ij} v_j / T)]
\]

The key advantage of the MFT approach is speed - stochastic updating is replaced by a set of deterministic equations. Another virtue is that one gets a transparent probabilistic interpretation of neuronic outputs since \( v_i = \langle s_i \rangle_T \). Also these equations are isomorphic to RC-equations and hence naturally map onto custom made hardware (see Section 7).

5.5 Mean Field Learning

Armed with the MFT approximation we now revisit the Boltzmann machine, where the origin of the time consumption is the stochastic computation of \( \langle s_i s_j \rangle_T \) etc. stochastically. From the definition of \( Z \) (eq. (5.15)) one has

\[
\langle s_i s_j \rangle_T = -T \frac{\partial \log Z}{\partial \omega_{ij}}
\]

11) In feed-back networks it is customary to denote the local field by \( u_i \) rather than \( a_i \), which is the feed-forward (fan-in) counterpart.
Using eq. (5.28) one can then evaluate \((s_is_j)_T\) directly in terms of the MFT variables \(v_i\)

\[
(s_is_j)_T = -v_iv_j + 2v_j \tanh(\sum_k \omega_{ik}v_k/T)
\]

\[
= v_iv_j
\]  

(5.32)

This means that the BZ updating equation takes the simple form

\[
\Delta \omega_{ij} = \eta(v_i^sv_j^s - v_iv_j)
\]

(5.33)

In summary this MFT algorithm follows the steps of fig. 5.8.

1. Initialize \(\omega_{ij}\) with \(\pm\) random values.
2. Repeat until \(\omega_{ij}\) has converged:
2.1 **Clamped phase.** The values of the input and output units of the network are clamped to a training pattern, and for a sequence of decreasing temperatures \(T_n > T_{n-1} > \ldots > T_0\), eqs. (5.29) are solved iteratively. At \(T = T_0\), \(v_iv_j\) is computed.
2.2 **Free phase.** The same procedure as in step 3, but this time the network runs freely or with only the input units clamped. Again \(v_iv_j\) is computed at \(T = T_0\).
2.3 Update \(\omega_{ij}\) according to

\[
\omega_{ij}(t+1) = \omega_{ij}(t) + \eta(v_i^sv_j^s - v_iv_j).
\]

Figure 5.8: A mean field learning algorithm.

The performance of this MFT learning algorithm is comparable with BZ and BP [42]. The expected speedup as compared to BZ has been verified [42]. In the limit of few output units it reduces to BP if the cross-entropy error (eq. (3.12)) is used for the latter [61, 87]. In associational memory applications it gives rise to capacities \(N_p \sim O(10 - 20)N_H\), where \(N_H\) is the number of hidden units [62]. This number should be compared with \(N_p \approx 0.14N_H\) for the Hopfield model.

6 Optimization Problems

Neural networks can also be used to find good approximate solutions to difficult combinatorial optimization problems [51, 52, 64, 65], which are NP-complete\(^\text{12}\). Exact solutions to these problems require state space exploration leading to \(n!\) or \(a^n\) computations for a system with \(n\) degrees of freedom. Different kind of heuristic methods are therefore often used to find reasonably good solutions. The ANN approach, which is based on feedback networks, falls within this category. It has advantages in terms of solution quality and a "fuzzy" interpretation of the answers through the MFT variables. Furthermore it is inherently parallel, facilitating implementations on concurrent processors and with MFT equations custom made hardware is straight-forward to design.

There are two families of ANN algorithms for optimization problems:

- "Pure" neural approach based on either binary [51] or multi-state [52] neurons with MFT equations for the dynamics.

---

\(^{12}\) A problem is NP if there is no known algorithm that solves it in polynomial time. NP-complete is a class of NP problems such that if one finds a polynomial solution to one of the members, all the others are solved as well.
— Deformable templates [66], where the neuronics degrees of freedom have been integrated out and one is left with coordinates for possible solutions.

The latter pathway is a winner for low-dimensional geometrical problems like the traveling salesman problem (TSP), whereas the former is the only possibility for high-dimensional problems like scheduling. The following sections describe both these approaches illustrated with suitable problems.

6.1 The Neural Approach
6.1.1 The Graph Bisection Problem

The "pure" neural approach is well illustrated with the graph bisection problem. This problem is defined as follows (see fig. 6.1a): Partition a set of N nodes with given connectivity into two halves such that the connectivity (cusize) between the two halves is minimized.

![Graph Bisection Problem](image)

Figure 6.1: (a) A graph bisection problem. (b) A K = 4 graph partition problem. (c) A N=4 TSP problem.

The problem is mapped onto the Hopfield energy function (cf. eq. (5.4)) by the following representation: For each node, assign a binary neuron \( s_i \) and for each pair of vertices \( s_i, s_j \) \( i \neq j \) we assign a value \( \omega_{ij} = 1 \) if they are connected, and \( \omega_{ij} = 0 \) if they are not. In terms of fig. 6.1a, we let \( s_i = \pm 1 \) represent whether node \( i \) is in the left or in the right position. With this notation,

\[
\omega_{ij} s_i s_j = \begin{cases} 
> 0 & \text{whenever } i \text{ and } j \text{ are in the same partition} \\
= 0 & \text{if } i \text{ and } j \text{ are not connected at all} \\
< 0 & \text{whenever } i \text{ and } j \text{ are in different partitions}
\end{cases}
\] (6.1)

Minimization of the energy function (5.4) will maximize the connections within a partition while minimizing the connections between partitions, with the result that all nodes are forced into one partition. Hence we must add a "constraint term" to the right hand side of eq. (5.4) that penalizes situations where the nodes are not equally partitioned. We note that \( \sum s_i = 0 \) when the partitions are balanced and a term proportional to \( (\sum s_i)^2 \) will subsequently increase the energy whenever the partition is unbalanced. Our energy function for graph bisection thus takes the form:

\[
E = -\frac{1}{2} \sum_{ij} \omega_{ij} s_i s_j + \frac{\alpha}{2} (\sum_i s_i)^2
\] (6.2)

where the Lagrange multiplier (imbalance parameter) \( \alpha \) sets the relative strength between the cutsize and the balancing term. This balancing term represents a global constraint. The generic form of eq. (6.2) is

\[
E = \text{"cost"} + \text{"global constraint"}
\] (6.3)
which is typical when casting "difficult" optimization problems onto neural networks. The origin of the difficulty is very transparent here; the problem is frustrated in the sense that the two constraints ("cost" and "global constraint") are competing with each other with the appearance of many local minima. (cf. the discussion of spin-glasses above). These local minima can to a large extent be avoided by applying the MFT technique to eq. (6.2) yielding

\[ v_i = \tanh\left( \sum_j \left( \omega_{ij} - \alpha \right) v_j / T \right) \]  

(6.4)

The generic form of the energy function in eq. (6.2) is very different from a more standard heuristic treatment of the optimization problem. For example in the case of graph bisection one typically starts in a configuration where the nodes are equally partitioned and then proceeds by swapping pairs subject to some acceptance criteria. The constraint of equal partition is respected throughout the updating process. This is in sharp contrast to neural network techniques (eq. (6.2)), where the constraints are implemented in a "soft" manner by a Lagrange multiplier. The final MFT solutions are therefore sometimes plagued with a minor imbalance, which is easily remedied by applying a greedy heuristic to the solutions [87].

Very good numerical results were obtained for the graph bisection problem in ref. [67] for problem sizes ranging from 20 to 2000. The quality of the solutions were comparable with those of the CPU demanding simulated annealing method. The time consumption is lower than any other known method. This holds even though the possibility of parallel execution has not been used. In fig. 6.2 we show typical evolutions of systems as functions of number of iterations, it illustrates well the "fuzziness" of the system; some \( v_i \)-values never converge to 1 or -1.

![Graph Bisection Examples](image_url)

*Figure 6.2: Evolution of \( v_i \) for a \( N=100 \) and \( N=500 \) graph bisection problem.*

The MFT equations (eq. (5.29)) can for these applications either be solved by iterations at a fixed low temperature \( T \) or by annealing in \( T \). Very good numerical results are obtained with this method [67]. The temperature is a free parameter of the system (in addition to \( \alpha \)). The system has two phases (cf. fig. 5.2); at large enough temperatures \( (T \to \infty) \) the system relaxes into the trivial fixed point \( v_i^{(0)} = 0 \). As the temperature is lowered a phase transition is passed at \( T = T_c \) and as \( T \to 0 \) fixed points \( v_i^{(*)} \) emerge representing a specific decision made as to the solution to the optimization problems in question.
The position of $T_c$, which depends on $\omega_{ij}$ and $\alpha$, can be estimated by expanding the sigmoid function (tanh) in a power series around $v_i^{(0)} = 0$ (see fig. 6.3). The fluctuations around $v_i^{(0)}$

$$v_i = v_i^{(0)} + \epsilon_i$$

satisfy

$$\epsilon_i = \frac{1}{T} \sum_j m_{ij} \epsilon_j$$

where $m_{ij} = \omega_{ij} - \alpha$. For synchronous updating it is clear that if one of the eigenvalues to $m/T$ in eq. (6.6) is $> 1$ in absolute value the solutions will wander away into the nonlinear regions. In the case of serial updating the philosophy is the same but the analysis slightly more complicated. We refer the reader to ref. [52] for a more detailed discussion. Finding the largest eigenvalue to $m$ could be computationally explosive by itself. Is there an approximate way of doing it? Yes, there is. Providing $\langle \omega_{ij} \rangle$ and the corresponding standard deviation $\sigma$ turns out to be sufficient for obtaining estimates within 10% of $T_c$. Also, this analysis is important for avoiding oscillatory behavior [68], which appears for eigenvalues $<-1$.

With this method of estimating $T_c$ in advance we thus have a "black box" reliable, parallelizable powerful algorithm for solving problems of this kind.

6.1.2 The Traveling Salesman Problem

When generalizing the graph bisection problem to graph partitioning (GP) the $N$ nodes are to be partitioned into $K$ sets, each with $N/K$ nodes, again with minimal cutsize (see fig. 6.1b). This requires introducing a second index for the neurons (neuron multiplexing)

$$s_{ia} = 0, 1$$

where the index $i$ denotes the node ($i = 1, ..., N$) and $a$ the set ($a = 1, ..., K$). The neuron $s_{ia}$ takes the value 1 or 0 depending on whether node $i$ belongs to set $a$ or not. The same structure is present in TSP (see fig. 6.1c), where $N$ cities are to visited exactly once each with a minimal total tour length. Let $s_{ia}$ be 1 if city $i$ has the $a$th tour number and let $d_{ij}$ be the distance between city $i$ and $j$. With neuron multiplexing the energy can be written as [51]

$$E = \sum_{ij} d_{ij} \sum_a s_{ia} s_{j(a+1)} + \frac{\beta}{2} \sum_i \sum_{a \neq b} s_{ia} s_{ib} + \frac{\alpha}{2} \sum_i (\sum_a s_{ia} - N)^2$$

where the first term measures the total distance of the tour, the second term penalizes situations where a city appears more than once in the tour and the third term ensures

13) We here use $\{0, 1\}$ notation in order to get a more convenient form of the energy function.
that all cities are visited at least once. Again mean field variables $v_{ia}$ can be defined and the corresponding MFT equations solved. It turns out that the results are extremely parameter sensitive [51, 52].

In ref. [52] an alternative encoding method, multi-state or $K$-valued neurons, was proposed. A Potts condition (see section 3.2.2) was imposed to account for

\[ \sum_a s_{ia} = 1 \]  

Thus for every $i$, $s_{ia}$ is one for only one value of $a$, and zero for the remaining values of $a$. So, the allowed values of the vector $\hat{s}_i = (s_{i1}, s_{i2}, \ldots, s_{iK})$ are the principal unit vectors $\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_K$ in an obvious vector notation (cf. sect. 3.2.2). The number of states available at every node is thereby reduced from $2^K$ to $K$, and technically we have a $K$-state Potts model at our hands. In fig. 3.6 we show the space of states at one node for the case $K = 3$.

The energy function of eq. (6.8) can now be rewritten, using the constraint of eq. (6.9), as

\[ E = \sum_{ij=1}^{N} d_{ij} \sum_{a=1}^{K} s_{ia} s_{ja(a+1)} = \frac{\beta}{2} \sum_{i=1}^{N} \sum_{a=1}^{K} s_{ia}^2 + \frac{\alpha}{2} \sum_{a} \left( \sum_{i} s_{ia} \right)^2 \]  

We next need MFT equations for Potts neurons (eq. (3.14)). These can be derived in a manner similar to the binary neuron case [52]. They read (cf. eq. (3.14))

\[ v_{ia} = \frac{e^{u_{ia}}}{\sum_b e^{u_{ib}}} \]  

with

\[ u_{ia} = \frac{\partial E}{\partial v_{ia}} \frac{1}{T} \]  

which for $K = 2$ gives reduces to a tanh-function as expected. This expression automatically satisfies the constraint $\sum_a v_{ia} = 1$. Thus, when iterating eq. (6.11), the mean field variables $v_{ia}$ will be forced to live in this $(K-1)$-dimensional subspace of the original $K$-dimensional unit hypercube, as shown in fig. 3.6 for the case $K = 3$.

As in the graph bisection case above, approximate values for $T_c$ can be found by linearizing eq. (6.11). This algorithm for TSP, which is of “black-box” nature, is summarized in fig. 6.4 (for graph partition problems the algorithmic steps are identical).

---

1. Choose problem $d_{ij}$
2. Find the approximate phase transition temperature by linearizing eq. (6.11).
   (for details see ref. [52])
3. Initialize the neurons $v_{ia}$ with $\pm$ random values.
4. Anneal until $\Sigma \equiv \frac{1}{N} \sum_i v_{ia}$ is equal to 0.9:
   4.1 $T_n = 0.9 \times T_{n-1}$;
   4.2 At each $T_n$ update $v_{ia}$;
   \[ v_{ia} = \frac{e^{u_{ia}}}{\sum_b e^{u_{ib}}} \cdot \]
5. After $\Sigma = 0.9$ is reached perform a greedy heuristics to account for possible imbalances or rule violations.

---

Figure 6.4: A Potts neurons algorithm for TSP.

14) This is of course not the only way of coding the problem. One could e.g. interchange the roles of the labels $i$ and $a$. 
This TSP algorithm performs very well in comparison with others — all solutions are within \( \sim 5\% \) from the simulated annealing average \([52, 71]\). These results are for random distribution of cities. For realistic more structured problems results are even better. The disadvantage of this ANN approach is that \( N^2 \) degrees of freedom are needed for \( N \) cities. We will next discuss an ANN related approach that avoids this problem of large number of degrees of freedom.

6.2 Deformable Templates

Let us denote the cities in the TSP by \( \bar{x}_i \) (see fig. 6.5). We are going to match these cities with template coordinates \( \bar{y}_a \) such that \( \sum_a |\bar{y}_a - \bar{y}_{a+1}| \) is minimized and that each \( \bar{x}_i \) is matched by at least one \( \bar{y}_a \). Define \( s_{ia} \) to be 1 if \( a \) is matched to \( i \) and 0 otherwise. The following energy expression then minimizes a valid tour

\[
E(s_{ia}, \bar{y}_a) = \sum_{ia} s_{ia} |\bar{x}_i - \bar{y}_a|^2 + \gamma \sum_a |\bar{y}_a - \bar{y}_{a+1}|^2
\]  

(6.13)

The Lagrange multiplier \( \gamma \) governs the relative strength between matching and tour length (cf. \( \alpha \) and \( \beta \) in the neuromorphic description above). Note that with eq. (6.13) the problem is parameterized in two ways — with the "neurons" \( s_{ia} \) and with the template coordinates \( \bar{y}_a \). The Boltzmann distribution for the energy in eq. (6.13) reads

\[
P(s_{ia}, \bar{y}_a, T) = \frac{e^{-E(s_{ia}, \bar{y}_a)}/T}{Z}
\]

(6.14)

with the partition function \( Z \) given by

\[
Z = \sum_{s_{ia}} \sum_{\bar{y}_a} e^{-E(s_{ia}, \bar{y}_a)}/T
\]

(6.15)

We can now define so-called marginal distributions by either integrating out the neuromorphic or the template coordinate degrees of freedom. If we choose the latter alternative we end up with a pure neuromorphic description of the problem similar to the one in the previous section. We now choose the former alternative. Performing the sum over \( s_{ia} \) and using some "tricks" (see e.g. refs \([69, 70]\)) one obtains the marginal distribution \( Z_M \) as

\[
Z_M = \sum_{\bar{y}_a} \prod_i (\sum_a e^{-|\bar{x}_i - \bar{y}_a|^2/T}) e^\gamma \sum_a |\bar{y}_a - \bar{y}_{a+1}|^2/T
\]

(6.16)

Only those configurations where \( s_{ia} = 1 \) for only one \( a \) for each \( i \) have been summed over. We rewrite eq. (6.16) as

\[
Z_M = \sum_{\bar{y}_a} e^{-E_{eff}(\bar{y}_a)}
\]

(6.17)

where the effective energy \( E_{eff} \) is given by

\[
E_{eff}(\bar{y}_a) = -T \sum_i \log(\sum_a e^{-|\bar{x}_i - \bar{y}_a|^2/T}) + \gamma \sum_a |\bar{y}_a - \bar{y}_{a+1}|^2/T
\]

(6.18)

Next we minimize \( E_{eff} \) with respect to \( \bar{y}_a \) using gradient descent

\[
\Delta \bar{y}_a = \eta [2 \sum_{ia} \bar{v}_{ia} (\bar{x}_i - \bar{y}_a) + \gamma (\bar{y}_{a+1} - 2\bar{y}_a + \bar{y}_{a-1})]
\]

(6.19)

where the Potts factor (cf. eq. (6.11)) \( \bar{v}_{ia} \) is given by

\[
\bar{v}_{ia} = \frac{e^{-|\bar{x}_i - \bar{y}_a|^2/T}}{\sum_b e^{-|\bar{x}_i - \bar{y}_b|^2/T}}
\]

(6.20)
Thus in contrast to the Potts description in the previous section, the logical units here are not dynamical degrees of freedom, the analog variables \( \vec{y}_a \) play that role instead.

How does this algorithm work? At high temperatures \( T \) the template cities are located on a closed string with an origin in the center of mass of the cities (see fig. 6.5). As \( T \) is lowered the first term in eq. (6.18) becomes more important and matching takes place.

![Diagrams](image)

(a) (b) (c) (d)

Figure 6.5: Development of the template “snake” from high to low temperatures.

The first term in eq. (6.18) is a sum of Gaussians around the templates with width \( T \). At high \( T \) these Gaussians each attract all cities. As \( T \) decreases they focus on a smaller set of cities and finally each \( \vec{y}_a \) tries to match one \( \vec{x}_i \).

This algorithm, which is summarized in fig. 6.6, produces very high quality solutions [71]. And, very importantly, an \( N \)-city problem only requires \( 2N \) degrees of freedom. For problems embedded in low dimensional spaces like TSP the template method is always to prefer over the pure neuronal (Potts) one. However, for high dimensional problems like scheduling [64, 65] one must use the Potts neural network.

The structure of the first term in eq. (6.19) is similar to the one for collective updating of self-organizing networks (eq. (4.12)). The templates \( \vec{y}_a \) play the role of Gaussian centers that are updated such that each of them will, at low temperatures (narrow peaks), match exactly one city. At very high temperatures they are attracted to (contain) all cities. The main difference between this approach and the self-organizing networks is the neighborhood function, which for this case is \( \Lambda = \vec{v}_{ia} \), i.e. each “weight” (template) is updated to some degree through the Potts updating equation (eq. (6.20)). At very low temperatures \( \vec{v}_{ia} \) approaches \( \delta(\vec{z}_i - \vec{y}_a) \) and the competitive winner-takes-all updating is retrieved.
1. Choose problem - \( \vec{x}_i \)'s.
2. Choose number of templates \( \vec{y}_a \), \( a=1,...,M \); \( M > N \).
3. Compute the center of gravity of the cities \( \vec{x}_i \) and displace it slightly with a random seed. Place templates \( \vec{y}_a \) with equal spacing on a circle around this center.
4. For a sequence of temperatures \( T_n \) do:
   4.1 \( T_n = 0.9 \times T_{n-1} \).
   4.2 At each \( T_n \) update templates \( \vec{y}_a \);
       \( \vec{y}_a(T_n) = \vec{y}_a(T_{n-1}) + \Delta \vec{y}_a \)
       where \( \Delta \vec{y}_a \) is given by eq. (6.19).

Figure 6.6: An elastic net algorithm for the TSP.

7 Hardware

As we have demonstrated, the ANN approach is very powerful with respect to performance and generality. It is also very appealing from the point of view of concurrency — it easily lends itself to parallel execution. Most applications to date have been of off-line nature and mainly consist in simulating ANN on serial computers, without utilizing the inherent parallelism in the equations. Exploiting this parallelism can proceed in three different ways:
1. Using a general purpose multi-processor system like a CRAY, connection machine, hypercube, transputer system etc..
2. Realizing the ANN in special purpose VLSI electronic hardware, digital or analog.
3. Realizing the ANN with a non-linear optical system.

ANN's are characterized by very elementary processing units and any special purpose hardware should capitalize as much as possible on this fact. Another major property of an ANN is the large degree of connectivity, which poses a real challenge for VLSI implementations. In optical computing, on the other hand, large connectivity is natural. Optical computing has however not yet matured into massive product lines (but is rapidly gaining momentum).

This section is far from complete and exhaustive. It is merely intended as bringing up a few pointers. We refer the interested reader to the cited work.

7.1 VLSI Implementations

VLSI implementations follow two different approaches; digital and analog. Artificial neurons are analog by nature so the latter approach is natural. Digital implementations are expected to give slower chips than analog ones, but substantial efforts have gone into the development of digital transistor implementations of ANN (for a review see e.g. [72]). In what follows we will stick to analog implementations. An example of an analog electronic neuron is shown in fig. 7.1a. It is particularly interesting to look at feed-back analog systems since they are so closely related to the MFT equations (eq. (5.29,5.30)), which are heavily used in optimization applications and in the MFT version of the Boltzmann machine.
Figure 7.1: (a) An analog electronic neuron. (b) A VLSI implementation of a fully connected network.

The MFT equations have a very important property from the VLSI implementation point of view. They are identical to the static limit of the resistance-capacitance (RC) charging equations of a circuit of analog amplifiers [50, 51]. Consider a circuit of nonlinear amplifiers that convert an input voltage \( u_i \) to an output voltage \( v_i = \tanh(u_i/T) \) and are connected with resistors \( \omega_{ij}^{-1} \) between amplifier \( i \) and \( j \) (see fig. 7.1b). Positive and negative\(^{15} \) \( \omega_{ij} \)-elements are implemented by having a pair of connections between the amplifiers; \( \omega_{ij}^{(+)} \) and \( \omega_{ij}^{(-)} \). Each amplifier is given two outputs, a normal \(( > 0 \) \) and an inverted one \(( < 0 \) \). The weights \( \omega_{ij}^{(+)} \) and \( \omega_{ij}^{(-)} \) are both positive but interpreted with different signs depending on the sign of the outgoing voltage from the amplifier. The steady-state circuit equations are then given by

\[
\frac{u_i}{R_i} = \sum_j \omega_{ij}(v_j - u_i) = 0
\] (7.1)

where \( R_i \) are input resistors for the amplifiers leading to reference ground. With the replacement

\[
\omega_{ij} \rightarrow \left( \frac{1}{R_i} - \sum_k \omega_{ik} \right)^{-1} \omega_{ij}
\] (7.2)

and substituting \( T \tanh^{-1}(v_i) \) for \( u_i \), eq. (7.1) is identical to eq. (5.30). This close mapping between feed-back neural models and VLSI circuitry is indeed very appealing.

Feed-back networks along these lines have been developed. In ref. [74] \( N=256 \) fully connected neurons were implemented with e-beam lithography (fixed resistors). In refs. [75, 76] transistors were used for the \( \omega_{ij} \) to make them modifiable. Ref. [75] also has on-chip learning using the local learning rule of the Boltzmann machine.

An analog feed-forward implementation with off-chip learning (software simulation) is commercially available (ETANN) [77].

7.2 Optical Implementations

Since light beams can cross each other, optical computing is well suited for parallel processing with high connectivity. A key element in ANN processing are matrix multiplications \( u_i = \sum_j \omega_{ij} v_j \), that are very natural to implement optically. To see this, consider

\(^{15} \) Resistors are of course always positive.
a set of light emitting diodes (LED) and a set of photodetectors (PD) with a spatial light modulator in between (see fig. 7.2). With appropriate lenses these devices can perform a matrix multiplication when the SLM is used to store the matrix elements $\omega_{ij}$. As in the VLSI case the connection strengths need to be multiplexed ($\omega_{ij}^{(+)}$ and $\omega_{ij}^{(-)}$) in order to account for negative weights. In fig. 7.2 is indicated how to solve the MFT-equations in a feed-back network by closing a circuit with an amplifier (tanh). Instead of a SLM, which is typically an electronically loaded liquid crystal, one can use a photorefractive crystal to represent $\omega_{ij}^{(+)}$ [73, 78, 79]. In this case the mask is prepared with a volume hologram using coherent light. With this technology one can store approximately $10^9$ weights per cm$^2$, which is smaller than the storage capacity in VLSI but without scaling limitations in the connectivity. The system is also slower than the VLSI alternative ($\mathcal{O}(\text{ms})$) but the very large scale interconnection capacity is hard to beat!

$$v_i = \tanh(u_i/T)$$

![Figure 7.2: A generic optical neural network.](image)

8 High Energy Physics Applications

High energy physics contains many challenging feature recognition problems ranging from off-line data analysis to low-level experimental triggers. Flavor tagging and Higgs detection are obvious examples. With powerful algorithms at our hands, signal-to-background ratios can be substantially reduced. In particular for the next generation of accelerators (LHS, SSC) the availability of algorithms that can be executed in real-time will be crucial. The event rate at these machines is of the order of one event per 10–100 ns. Another class of feature recognition problems is track finding. Again, with the high luminosity expected at LHS and SSC real-time track reconstruction would be advantageous, if not crucial.

It turns out that proof-of-concept for using artificial neural networks (ANN) in high energy physics can be established already in off-line analysis of data from existing accelerators. In the domain of pattern recognition; quark-gluon separation, heavy quark tagging and Higgs search, results have been achieved with feed-forward ANN that are superior to conventional approaches [43, 44, 81, 82, 83, 84, 85]. For function approximation tasks; like reconstructing the mass of a decayed particle from calorimeter information, feed-forward networks have outperformed standard methods [31]. Also promising approaches to use feed-back ANN for solving the optimization problem of track finding have been suggested [86, 87] and have recently been successfully confronted with realistic data [88]. Alternative ANN inspired algorithms based on deformable templates have also been successfully explored for tracking [70, 89].
8.1 Jet Identification

We here describe off-line applications; using ANN for discriminating quarks from gluons and heavy quark tagging.

8.1.1 Quark-gluon Separation

Predicting whether a jet of hadrons originates from a quark or a gluon is important from many perspectives. It can shed light on the hadronization mechanism. For example, experimental studies of the so-called string effect needs identification of the gluon jet. Also, a fairly precise identification of the gluon jet is required for establishing the existence of the 3-gluon coupling in $e^+e^-$ annihilation reactions. In refs. [81, 82, 83] the back-propagation (BP) algorithm for MLP networks was used to do the separation both in $e^+e^-$ reactions and in hadron-induced large $p_\perp$ processes.

Electron-positron reactions [81, 82]

In order not to reveal too much about the MC model dependent low momentum part of the jet, four-momenta $(\vec{p}_k, E_k)$ of the four leading particles in the jet were used as inputs to the network. The network was a three-layered MLP with 16 inputs, 10 hidden units and one output representing the quark/gluon option (quark → 0 and gluon → 1). Training and test sets were generated at 2 different energies (92 GeV and 29 GeV) with 3 different MC generators; JETSET [90], ARIADNE [91] and HERWIG [92]. The inputs were either single jets defined by the LUCLUS clustering algorithm in JETSET [90] or entire 3-jet events (in which case the network was enlarged). After training was completed the network was tested with a middle point success criteria where $> 0.5$ for the output node was interpreted as a gluon jet and $< 0.5$ as a quark jet. For the 3-jet events, winner-takes-all criteria for three output units was used to determine the gluon jet.

On the average the network was able to correctly classify 85% of the test set jets. The MC model independence of the results were demonstrated by training on MC data generated by one model and tested on MC data from another. Almost no deterioration in performance was observed. Runs where detector acceptance effects were included showed only $O(2\%)$ degradation.

The QCD matrix element suppresses gluon jet production as $1/E_{\text{gluon}}$. A fair part of the ANN discrimination originates from this property. In order to factor out this matrix element dependence from the intrinsic differences between quark and gluon jets one should train different networks with quark and gluon jets in different energy intervals and combine the answers with the appropriate matrix element predictions. In ref. [84] such a procedure was pursued using more kinematical information than in refs. [81, 82]. The classification power then increases to 92$\%$\(^{16}\).

Large $p_\perp$ processes [43]

In this case the network gets no lead from QCD matrix element information, since the kinematics of the incoming quarks is unknown. Hence we expect a lower classification performance. Another difference is that in hadron-hadron collisions the momenta and energies of the produced hadrons are available in a “raw” form in terms of towers in a calorimeter representing the transverse energies $E_\perp$. In ref. [83] a set of $p\bar{p}$ events at 630 GeV were generated with the PYTHIA MC [93]. The transverse energy of the fragmentation products was mapped onto a calorimeter with a granularity of $\Delta\eta = 0.20$ in pseudorapidity ($\approx$ longitudinal velocity) and $\Delta\phi = 0.26$ in azimuth angle. The setup corresponded to the UA2 calorimeter at CERN and had a complete coverage in $\phi$ and extended up to $|\eta| \leq 2$. The size of a jet was defined as a $7 \times 7$ matrix in the calorimeter, approximately corresponding to a cone of radius 0.8 in $(\eta, \phi)$-space. The calorimeter information was presented to the network as follows: Take the $E_\perp$ of the leading cell in the $7 \times 7$ matrix

\(^{16}\) Including more kinematical information than the four leading hadrons might destroy the MC model-independence of the approach in refs. [81, 82].
and assign it to the first node \( x_1 \). Assign the \( \eta \) and \( \phi \) coordinates relative to the center of the jet to \( x_2 \) and \( x_3 \) respectively. Then take the second leading cell and assign its \( E_\perp, \eta \) and \( \phi \) to \( x_4, x_5 \) and \( x_6 \) and so on for the largest 15 cells. This corresponds to 45 input nodes. The reason for choosing this representation of the input data rather than the \( 7 \times 7 \) cells directly is that in this way invariances of the data is more efficiently incorporated. A three-layered MLP was used with 45 inputs, 10 hidden and one output unit. After training, the network correctly classifies \( 70 - 72\% \) of the jets using the 0.5-criteria (midpoint) as above. However, a simple midpoint performance is not so interesting. Instead one should vary the cut of the output node and choose a value corresponding to an optimal efficiency and signal-to-background ratio. In fig. 8.1 the increase in signal-to-background ratio as a function of the signal efficiency is shown. In ref. [83] a similar encoding was successfully used for separating jets stemming from the intermediate vector boson \( W \) from those originating from QCD collision processes. Such a network is able to reduce the QCD background to the process \( W/Z^0 \rightarrow \text{jets} \) by factors 20-30.

![Figure 8.1: The signal-to-background increase versus the efficiency for a neural network quark trigger.](image)

### 8.1.2 Heavy Quark Jet Tagging

So-called heavy quarks (\( c \) and \( b \)) are produced at high energies. In contrast to \( u \) - and \( d \)-quarks they are unstable and decay weakly, emitting leptons. The conventional way of identifying heavy quarks in \( e^+e^- \) reactions is either through leptonic tagging or secondary vertex with efficiency/purity levels of approximately \( (5\% / 95\%) \) and \( (25\% / 80\%) \) respectively. We trained a network to identify \( b \)-quarks [82], entirely based upon hadronic information. As input variables was used the total jet energy and momentum along with the energy and direction for the 6 leading particles in the jet. Again a three-layered MLP was used, with 20 input units, 10 hidden and a single output neuron \( (b \rightarrow 1 \) and non-\( b \rightarrow 0) \). As in the large \( p_T \) application above the option of varying the cut on the output node can be used to select the desired efficiency vs. purity performance. The network is then able to produce efficiency/purity numbers comparable with what is expected from vertex detectors. This remarkable result implies that a general purpose hadronic detector could be very efficient also for heavy flavor tagging. Even better efficiency/purity ratios were obtained in subsequent work [94] by preprocessing the kinematic variables in terms of different shape variables. In fig. 8.2 the distribution of events for the output node is shown from [94]. It is clear from this figure how choosing the output threshold governs the efficiency/purity ratio.

It is illuminating to study heavy quark tagging in a self-organizing feature map. In ref. [43] such a network consisting of a plane of \( 7 \times 7 \) feature nodes was used to disentangle \( b \), \( c \) - and light (\( uds \)) quarks. The input layer has 12 nodes, corresponding to \( p_z \) and \( p_\perp \).
for the 6 leading particles in the jet. The resulting distributions over the feature nodes are shown in fig. 8.3. In fig. 8.4a the mapping is shown in terms of dominating quarks. Not surprisingly the two extremes in terms of quark masses occupy two distinct areas separated by the c-quarks. It is very interesting to inspect the corresponding weight vectors $\bar{w}_j$. The nodes in the lower right corner of fig. 8.4a have $\bar{w}_j$'s with an even distribution of momenta (see fig. 8.4b), which is exactly what one expects for b-quarks since they decay more isotropically.

The neighborhood size $\lambda$ was only allowed to decrease down to 1 for this map, keeping some "stiffness" in the network. The resulting map (fig. 8.4b) is thus an approximate plane in the dominant (the largest variances) directions of the inputs. In this case the plane is oriented along the $p_z$ for the first and second particles in the jet.

Figure 8.2: Output distributions for b (full line) and non-b (dashed line) hadronic jets (from ref. [84]).

Figure 8.3: Distribution of b-quarks (a), c-quarks (b) and uds-quarks (c) over the self-organized 7 x 7 feature nodes.
Figure 8.4: (a) The resulting map for $uds$, $c$, and $b$-quarks. The shading indicates the dominant flavor for the units. The units are numbered as in fig. 8.3. (b) The weight vectors, corresponding to $\vec{p}$ for the 4 leading hadrons, for the corner units; $(7,1)$ is $b$-sensitive and $(1,7)$ is $uds$-sensitive. The $p_\perp$ component has been multiplied by a factor of 5 relative to the $p_z$ component.

8.2 Mass Reconstruction

When hunting for new particles or resonances one often encounters the problem of computing invariant masses of expected decay products. For example, in the case of the intermediate vector boson $W$ (produced in $\bar{p}p$ collisions) in its hadronic decay channel, $W \rightarrow q\bar{q} \rightarrow$ hadrons, $M_W$ is reconstructed relativistically from the momenta and assumed masses of the produced hadrons. The problem here is that the $q$ and $\bar{q}$ jets are not the only hadrons in the collision - there are also remnants from projectile hadrons. An additional complication is that the $q$ and/or $\bar{q}$ jets can give rise to additional jets through bremsstrahlung. Identifying the appropriate jets is thus crucial for a good reconstruction of the mass. The "standard" procedure [80] for doing this is by sweeping through the calorimeter with a "window" of a certain $(\phi, \eta)$ size. The two windows with largest total $E_T$ are selected as containing the two jets and the hadrons in these jets are used to compute $M_W$.

The neural network approach to this problem is as follows [31]: This is an example of an function approximation task and a linear rather than non-linear output node is used for the answer ($M_W$). As inputs the 30 largest towers from the calorimeter are used (cf. quark/gluon separation above) together with the total $E_\perp$. A network with two hidden layers with 36 and 15 hidden units respectively is trained with the BP algorithm. As training set MC generated data with a flat distribution of $M_W$ in the range [50,150] GeV is used. When tested on "real" data in terms of MC generated realistic $W$-masses the ANN approach produces a distribution which is more narrow and symmetric than the one using conventional methods (see fig. 8.5). The main reason why the ANN method does better than the conventional method is that it captures the gluon bremsstrahlung tails well.

8.3 Track Finding

The track finding problem is to construct smooth curves through a set of signal points subject to application specific requirements. Typical applications are determination of moving target trajectories and tracks in high energy physics experiments. Common for these applications is the existence of a set of $N$ signal points. The task is to connect the $N$ signal points into continuous smooth tracks.
8.3.1 The Neural Approach

In refs. [34, 86] binary neurons $s_{ij}$ were introduced to represent whether the decision is to connect points $i$ with $j$ or not. Consider an energy function of the form

$$E = E^{(\text{cost})} + E^{(\text{constr})}$$  \hspace{1cm} (8.1)

where the "cost" part $E^{(\text{cost})}$ is chosen such that short adjacent segments with small relative angles are favored. In ref. [87] the following choice was made

$$E^{(\text{cost})} = \frac{1}{2} \sum_{ijkl} \delta_{jk} \frac{\cos^m \theta_{ijkl}}{r_{ij} + r_{jl}} s_{ij}s_{kl}$$  \hspace{1cm} (8.2)

where $m$ is an odd integer and the line segments $r_{ij}$ and angles $\theta_{ijkl}$ are defined in fig. 8.6.

![Segment definition](image)

Figure 8.6: Definition of segment lengths $r_{ij}$ and angles $\theta_{ijkl}$ between segments.

The "constraint" term in eq. (8.1) has two parts $E_1^{(\text{constr})}$ and $E_2^{(\text{constr})}$, where $E_1^{(\text{constr})}$ takes care of the requirement that there should be no bifurcating tracks

$$E_1^{(\text{constr})} = \frac{\alpha}{2} \left( \sum_{ik} s_{ij}s_{kj} + \sum_{jl} s_{ij}s_{il} \right)$$  \hspace{1cm} (8.3)

The other part, $E_2^{(\text{constr})}$, ensures that the number of neurons that are "on" roughly equals the number of signals $N$. It takes the form

$$E_2^{(\text{constr})} = \frac{\beta}{2} \left( \sum_{ij} s_{ij} - N \right)^2$$  \hspace{1cm} (8.4)
In eqs. (8.3, 8.4) \( \alpha \) and \( \beta \) are Lagrange multipliers. The MFT equations

\[
v_{ij} = \frac{1}{2} \left[ 1 + \tanh \left( -\frac{\partial E}{\partial v_{ij}} \frac{1}{T} \right) \right]
\]

(8.5)

are employed to minimize \( E \). In refs. [86, 87] these equations produced good solutions to moderately sized problems with appropriate choice of parameters (\( \alpha, \beta \) and \( T \)). In figs. 8.7 and 8.8 we show a typical evolution of the solutions and the energy behavior with respect to the number of iterations. There are in principle \( N^3 \) operations to be carried out at each iteration. However, due to the local nature of most track finding problems, this number can be substantially reduced; neurons need only to be defined within an interaction radius \( R_c \). With on the average \( m \) "active" partners within \( R_c \) one needs only \( O(Nm^2) \) computations.

![Figure 8.7: Segments with \( v_{ij} > 0.1 \) at different evolution stages of the MFT equations.](image)

![Figure 8.8: Energy as a function of the number of updatings for the problem in fig. 8.7.](image)

In ref. [88] the number of active neurons was further reduced under realistic circumstances using data from the cylindrical ALEPH TPC detector at CERN. Only neurons connecting points satisfying the following cuts were included:

- \( \Delta \phi < 0.15 \) rad
- \( \Delta \theta < 0.10 \) rad
- difference in pad-row\(^{17} \) < 4

\(^{17}\) In a cylindrical detector concentrical pad-rows around the origin detect the signals.
Also only those neurons corresponding to segments pointing inwards towards the origin exist. In fig. 8.9a segments corresponding to a $\mathcal{Z}^0 \rightarrow \text{hadrons}$ event is shown prior to convergence. In fig. 8.9b the surviving $s_{ij}=1$ neurons are shown for the same event. In general the quality of the solutions are very good. The efficiency of the ANN algorithm is around 99% which is comparable with the conventional tracking method used in ALEPH (99.7%) [88]. In ref. [88] the authors also perform simulation studies with track densities corresponding to LHC and SSC detectors ($\sim 200$ tracks/event). Fig. 8.10 shows the time consumption of the ANN algorithm and the conventional method [88].

![Figure 8.9: (a) Active neuron segments for a real $\mathcal{Z}^0 \rightarrow \text{hadrons}$ event (x-y projection) from ref. [89]. (b) Final surviving $s_{ij}=1$ neurons for the same event.](image)

Figure 8.10: Execution time as a function of the number of tracks (from ref. [89]).

Two things emerge from fig. 8.10: One is that the ANN method scales better with the number of tracks than the conventional method. The other is that approximately 60% of the ANN time consumption is due to initialization of the network; computing weights from the observed coordinates.

A few variations of the neuronic approach are possible:

- If one knew the number of tracks in advance from e.g. histogramming (see next section) one could have chosen a different neural encoding where a neuron $s_{ia}$ is “on” if point $i$ is assigned to track $a$.

- One of the soft constraints in eq. (8.3) could have been realized in a “hard” way with Potts encoding, in which case one has $\sum_j s_{ij} = 1$.

Both of these items will be involved in development of the deformable templates method.
in the next section.

8.3.2 The Deformable Templates Approach

Even though the ANN approach above seems to work very well it may not be the optimal way to proceed for the high energy physics track finding problem:

- It only solves the combinatorial optimization part of the problem; assigns signals to tracks. In reality one also needs to know the momenta corresponding the tracks. In the neural approach one then has to augment the algorithm with some fitting procedure. It would be nice to have an algorithm that does both things simultaneously.

- The neural approach is presumably more powerful than what is needed for this problem. The parametric form of the tracks are known in advance — helices — but the network is given no prior information about this. The fact that the problem is fairly easy to solve for the ANN algorithm is reflected in a very smooth phase transition (see fig. 8.8). However, for other applications with no prior knowledge of the parametric form of the tracks, the very versatile ANN approach is the way to go.

- The number of degrees of freedom needed to solve the $N$ signal problem is large even with the connectivity restrictions imposed in refs. [87, 88]. For a problem with $N$ signals and $M$ tracks one should only need $N \times M$ degrees of freedom.

- As demonstrated in ref. [89] the neural approach is somewhat sensitive to noise. Again with prior knowledge of the parametric form one should be more robust with respect to noise.

All of these issues can be accounted for in the deformable templates (elastic nets) approach [70] discussed above for the TSP. A similar approach was independently pursued in ref. [89]. The strategy is to match the observed events to simple parameterized models where the form of the models contains a priori knowledge about the possible tracks — circles passing through the origin (the collision point). In addition, this formulation allows for some data points, hopefully those corresponding to sensor noise, to be unmatched. The mechanism involved is closely related to redescending $M$-estimators used in Robust Statistics [95].

Let us for simplicity limit the discussion to two dimensions, where $\theta_a$ and $\kappa_a$ are the emission angle and the curvature for the $a^{th}$ track. A Hough transform [96] is used to give initial conditions for the templates and to specify the number of templates required. Hough transforms are essentially variants of “histogramming” or “binning” techniques which have previously been applied to particle tracking.

Assume that we are armed with an initial set of $M$ deformable templates with coordinates $(\theta_a, \kappa_a)$ from the Hough transform. A fitness measure is defined between the measurement points and the deformable templates as (cf. eq. (6.13))

$$ E(s_{ia}, \theta_a, \kappa_a) = \sum_{i,a} s_{ia} M(\theta_a, \kappa_a, \bar{z}_i) + \lambda \sum_i (\sum_a s_{ia} - 1)^2, $$

(8.6)

where:

- $\bar{z}_i$ labels the positions of the measurement points ($i = 1, ..., N$).
- $\theta_a$ and $\kappa_a$ are the angle of orientation at the origin and the curvature of the circles with $a = 1, ..., M$ indexing the circles.
- $M(\theta_a, \kappa_a, \bar{z}_i)$, which we abbreviate to $M_{ia}$, is a measure of distance between the $i^{th}$ point and the $a^{th}$ circle.
- $s_{ia}$ is a binary decision unit (neuron) such that $s_{ia} = 1$ if the $a^{th}$ circle goes through the $i^{th}$ point and zero otherwise.

We want to minimize $E(s_{ia}, \theta_a, \kappa_a)$ with respect to $(s_{ia}, \theta_a, \kappa_a)$, subject to the global constraint that each point is either matched to a unique circle or not matched. More precisely; given $i$ there exist a unique $a$ such that $s_{ia} = 1$. The second term in eq. (8.6) imposes a penalty $\lambda$ if a specific point is unmatched to any circle.
The Boltzmann distribution for $E(s_{ia}, \theta_{a}, \kappa_{a})$ in eq. (8.6) reads

$$P(s_{ia}, \theta_{a}, \kappa_{a}) = \frac{e^{-E(s_{ia}, \theta_{a}, \kappa_{a})/T}}{Z}$$

(8.7)

As in the TSP case marginal distributions $Z_{M}$ are obtained by integrating out the neuronic degrees $(s_{ia})$ of freedom. One gets

$$Z_{M} = e^{-E_{eff}(\theta_{a}, \kappa_{a})/T}$$

(8.8)

where the effective energy $E_{eff}$ is given by

$$E_{eff}(\theta_{a}, \kappa_{a}) = -T \sum_{i} \log(e^{-\lambda/T} + \sum_{a} e^{-M_{ia}/T})$$

(8.9)

Gradient descent on eq. (8.9) gives

$$\frac{d\theta_{a}}{dt} = -\frac{\partial E_{eff}}{\partial \theta_{a}} = -\sum_{i} \frac{e^{-M_{ia}/T}}{e^{-\lambda/T} + \sum_{b} e^{-M_{ib}/T}} \frac{\partial M_{ia}}{\partial \theta_{a}}$$

(8.10)

$$\frac{d\kappa_{a}}{dt} = -\frac{\partial E_{eff}}{\partial \kappa_{a}} = -\sum_{i} \frac{e^{-M_{ia}/T}}{e^{-\lambda/T} + \sum_{b} e^{-M_{ib}/T}} \frac{\partial M_{ia}}{\partial \kappa_{a}}$$

(8.11)

These equations again have the interesting structure that decision elements are mixed with parameter fitting parts.

In ref. [70] it is shown how the Hough transform is a $T \to 0$ and $\lambda \to 0$ limit of eq. (8.6). In ref. [70] it is discussed how this approach is related to that of ref. [89], where also Hough transforms are used to specify the number of tracks and initial parameters values. The template "arms" corresponding to the different tracks are fitted to the data in a serial manner one by one; one elastic net is used per track and an explicit repulsive force is introduced between the tracks to ensure this. The approach here is somewhat different. First of all the elastic net consists of many "arms" finding the solution simultaneously. Repulsive forces between the tracks is implicitly present through the winner-takes-all mechanism of eqs. (8.10, 8.11).

The approach of ref. [70] has been explored with simulated DELPHI TPC events with encouraging results (see fig. 8.11) and this algorithm is presently being investigated in depth for a variety of applications [97].

Figure 8.11: (a) Simulated data from DELPHI TPC. (b) Result from hybrid Hough/deformable template system.

Most results reported in this section [81, 82, 83, 43] were obtained using the F77 subroutine package JETNET [99].

Ongoing projects [98] include reproducing the theoretical "QCD calorimeter", given the experimentally measured hadronic and electromagnetic showers, i.e. finding the inverse
to the QCD cascading, fragmentation and detector simulation processes. Another challenge [98] is to improve the signal/noise ratio for separating $H^0 \rightarrow W^+W^-$ at LHC/SSC energies where one of the $W$ bosons decays into hadronic jets from a background consisting of $W + \text{jets}$, $t\overline{t} \rightarrow W + \text{jets}$ and $b\overline{b} \rightarrow l + \text{jets}$.

9 Summary

9.1 The Neural Network Approach — What is New?

The question that immediately arises is to what extent the ANN approaches really represent something new or whether they are just conventional methods in new "clothes". We discuss this issue separately for pattern recognition and optimization problem applications.

Feature Recognition

The most commonly used "conventional" method in this application field is discriminant analysis, where a discriminant function $\mathcal{F}(\mathbf{x}^{(p)})$ is formed such that

$$
\mathcal{F}(\mathbf{x}^{(p)}) = \begin{cases} 
> 0 & \text{if } \mathbf{x}^{(p)} \in \Omega_1 \\
< 0 & \text{if } \mathbf{x}^{(p)} \in \Omega_2
\end{cases}
$$

for the case of two classes $\Omega_1$ and $\Omega_2$. A summed squared error (cf. eq. (3.1)) is minimized with respect to the parameters occurring in $\mathcal{F}$. Typically a linear function for $\mathcal{F}$ is assumed. The supervised MLP, being able to approximate any continuous function for $\mathcal{F}$, subsequently represents a generalization of this discriminant procedure. The MLP is a "black-box" method that also has the advantage of being straightforward to realize in special purpose hardware.

We mentioned in section 4 that competitive the self-organizing vector quantization is identical to the adaptive $k$-means-clustering method. The novel part here is the collective updating variants; topological ordering that makes visualization easy by dimensional reduction onto a space with defined local topology, and soft Potts-type updating, which facilitates to disentangle difficult regions.

Optimization Problems

Optimization is the least explored ANN application area to date, and it is also the one where the gap to conventional methods is largest. By mapping these problems onto feed-back networks one is able to utilize the powerful MFT approximation; which is very efficient for avoiding local minima and leads to a set of deterministic updating equations (in contrast to stochastic methods). Furthermore, the MFT equations are isomorphic to VLSI RC-equations, making the hardware implementation straightforward. The use of self-organizing (or templates) approaches to combinatorial optimization problems is also new.

9.2 Other Algorithms and Applications

These lectures have not been complete with respect to covering all application areas and algorithms. We have confined ourselves to the ones displaying the basic concepts well. For example processing of time-sequenced data (e.g. speech) was not covered, neither was the recurrent back-propagation algorithm [53], which is intimately related to the MFT learning procedure. Also various hybrid schemes exist, among which one should mention the radial basis function approach in ref. [6]. There a set of self-organizing feature nodes (with different widths) are first used to cluster the data, followed by an addition of output nodes and BP learning. We refer to e.g. ref. [1] for treatment of the issues omitted here.

9.3 Finite Bit Precision Requirements

Ultimately, these ANN applications should be implemented in VLSI hardware, in which case one important question arises: To what extent are the solutions sensitive to
the bit precision required for amplifiers and weights? This of course requires that both learning and testing is performed at the same level of precision. The option of cutting the precision exists in JETNET [99], and for quark triggering applications, it seems that 2 or 3 bits (except for the sign bit) in both amplifiers and weights are enough to retain good performance [98].

9.4 Outlook for High Energy Physics Applications

We have here demonstrated the power of the neural network paradigm for high energy physics applications. Needless to say the ANN approach looks very promising. The technology is certainly here to stay.

The neural network method is very efficient for extracting features in hadronic data. World record performance can be obtained for quark/gluon separation. With respect to heavy quark tagging the results are in parity with those expected from a vertex detector. A similar network is also able to reduce the QCD background to $W/Z^0 \rightarrow jets$ by a factor 20–30.

The neural network approach is in general very noise- and damage-resistant. Hence it is suitable for high energy physics experiments where various parts of a detector may malfunction. Also, with its inherent concurrency and simple structure, fast execution custom made hardware could be an asset for real-time triggering. Such systems are already underway [100].

In addition to feature recognition ANN will very likely also play an important role in track finding since feed-back networks easily lend themselves to real-time hardware implementations.

Acknowledgments:

We would like to thank L. Gislén, A. Nilsson and M. Ohlsson for valuable comments on the manuscript. These set of lectures benefitted from the nice atmosphere provided by the CERN School of computing organization and its very vital student body. We also thank DEC Sweden for providing the school with ample and up-to-date computer equipment for the hands-on exercises. In particular we thank Jan Andersson for installing the system.
REFERENCES


K. Hornik, M. Stinchcombe and H. White, "Multilayer Feedforward Networks Are Universal Approximators", Neural Networks 2, 359 (1989);
[34] C. Peterson and E. Hartman, "Explorations of the Mean Field Theory Learning Algorithm", Neural Networks 2 475 (1989).
[38] T. Kohonen, "Self Organized Formation of Topologically Correct Feature Maps", Biological Cybernetics 43, 59 (1982);


An Artificial Intelligence Approach to Data Analysis

K.-H. Becks and A. Hemker

Physics Department, University of Wuppertal, Wuppertal, Germany

Abstract

The title of my lectures was “Artificial Intelligence in Particle Physics”. As we presented a similar lecture at the 1989 CERN School of Computing, this article discusses only new aspects and further developments not yet covered that time. For a better understanding of this paper the reader should be familiar with Ref [1]. Within this contribution we shall discuss a possibly new method for the physics analysis task1). We shall see that neither a conventional expert system alone nor a chain of artificial neural networks can solve such a complex task. We have to use both techniques: Connectionist and parallel methods for pre-processing of data delivered from the detector systems of an experiment, and symbolic and sequential methods for performing the physics analysis step. This is the same way human beings proceed: fast condensing of input data into higher level information and performing logical inferences afterwards. For the second part of the task, the analysis step (our main subject of interest), we tried several problem solving methods derived from nature which are briefly described with actual references. The architecture of a possible system is sketched at the end. We believe that we have to explore the new techniques developed in the fields of neuro-informatics and artificial intelligence in order to examine possible advantages for our applications.

1 Introduction

Present knowledge based systems are not really intelligent. Fundamental deficiencies of the symbol processing approach prevent a tight coupling of expert systems to open environments in order to solve tasks such as data analysis or control. But, the core of intelligence lies in the dynamic interaction with real world processes.

Connectionist models used as highly parallel, adaptive, and fault tolerant interfaces can partly compensate the weakness of the symbolic paradigm of artificial intelligence without a full substitution [2]. Not only artificial neural networks but dynamical systems in general can lead in cooperation with inferential working components to a “more intelligent” expert system technology [3, 4]. We shall sketch a hybrid system consisting of a chain of neural networks for pre-processing of data from a high energy physics experiment and a “knowledge directed” genetic algorithm for performing the physical analysis.

2 The Application

The new electron positron collider LEP at CERN has started operation in July 1989. Experiments have been constructed to test the predictions of the standard model of weak interactions and models beyond that theory. In the collider ring, electrons and positrons are accelerated in opposite directions. Each time an electron collides with a positron many new and partially short-living particles are produced. Most of the particles can be identified with large detector systems, one of which is the DELPHI detector [5].

1) This work is supported by the “KI-Verbund NRW”, founded by the Ministry for Science and Research of North Rhine Westfalia.
The event analysis task is complicated by the indeterminism of the underlying physical processes (hard QCD effects, fragmentation, decays) and by the limited resolutions and efficiencies of the different detector systems. In addition, the high amount of data of about 100 KBytes at a recording rate of several Hertz sets tight time limits.

The so far used data analysis techniques are conservative and manifest themselves in a huge number of FORTRAN routines. The large amount of data is reduced to only a few physical, statistical, and geometrical values which are sensitive to the physics question under investigation. Data reduction in conjunction with a loss of information as well as flat inferences allow no safe conclusions on the basis of a few events.

On the other hand, a more advanced system which would incorporate methods of neuro-informatics and artificial intelligence could lead to a more effective and more qualitative analysis method. For such a system we would need an interface between the environment, i.e. the detectors, and the subsequent analysis steps to compress very quickly the incoming data into higher level information. The task such a pre-processing unit has to perform is selection of presumably good events (trigger), geometrical reconstruction of particle tracks, identification of particles, and detection and correction of erroneous data. We have to deal with typical problems of feature detection, pattern completion, and classification which have at least partly to be solved in real time. The pre-processing unit has to reduce the amount of data by filtering, compression, and interpretation without loss of relevant information. In addition, this pre-processing unit could be used to infer the state of the detector systems from the input data. A feedback component would be desirable to hold the detectors in a well defined equilibrium. A possible malfunction of a part of a detector could be automatically corrected by such a calibration component. By the described tight coupling it would be possible to react to small deviations and to inhibit serious detector problems.

The following physical analysis step then tries to infer (according to specific physical models) the underlying physical processes for instance in the annihilation of $e^+e^-$ starting from the observable stable hadrons. The different physical processes (particle decays, fragmentation, etc.) can be represented by substitution rules of a grammar. If the stable particles are regarded as terminal symbols and the initial $e^+e^-$ state as start symbol, then the reconstruction task can be interpreted as a word problem of a context free grammar. Besides the question of how to create a word (event) by a grammar (physical model) one is mainly interested in the inference tree (physical processes / intermediate states) which leads to the observed event.

An additional problem appears by the fact that not the word itself but only word fragments are known. This resembles a word puzzle with a randomly mixed up set of letters from which we have to compose the original word. The searched word has to be syntactically inferable from the substitution rules and in addition it has to be semantically (physically) sensible. For a typical DELPHI event at the $Z^0$ mass, the number of possible reconstruction steps is bigger than $10^{22}$. As the parsing steps have different physical quality and the number of applicable rules is limited by the particles' energy, not all regions of the problem space have to be explored. Within this picture the reconstruction task is an optimization task in a reduced but nevertheless extremely complex problem space.

For solving such problems we either need a huge amount of computing power or we have to use meta-knowledge to focus the search direction to the goal state. But, not enough heuristic knowledge for the construction of an efficient and deterministic search algorithm exists. As we have access to a multiprocessor machine, to a Connection Machine CM2 [6], installed at our university, we started to implement on this SIMD architecture a problem solving method which does not need the full information about the problem domain, but which is nevertheless able to perform model-based inferences.
3 Requirements

Similar requests as those described in the last chapter are found in such different application domains as planning, recognition and understanding of natural language, automated manufacturing, and image processing. For such applications one typically needs systems which have to act in an open and dynamical environment. The environment sends sensory information and is able to react to manipulations from the outside. As one has to solve cognitive tasks of different rank, one needs systems which display intelligence on several levels. In the event analysis task one has on the one hand to manipulate and to manage a big amount of incoming data, a more perceptive task, and on the other hand to implement a highly complex problem solving procedure to perform a physical analysis.

3.1 “Intelligent” Problem Solving Methods

What are the characteristics of systems which deal with a broad spectrum of cognitive performances? Although we do not try to answer the question of what intelligence is, we can nevertheless substantiate some relations: intelligence, parallel processing, distributed knowledge representation, self organization, and dynamical interaction with real world processes are conceptions which are tightly coupled and which stipulate each other. Intelligent behaviour manifests itself not only in the manipulation of a limited amount of symbols in a closed system, as can be seen by insufficiencies of today’s expert system technology. The kernel of intelligence rather then lies in the dynamical interaction with real world problems. Intelligence can prove itself if a system is able to perform well in a complex environment as for instance in our surrounding world. Embedding such a system into a complex environment needs parallel processing to handle the big amount of information. In order to avoid limitations of processing and storage, the knowledge has to be distributed. Highly parallel and distributed systems can no longer be programmed. Because of their complexity, they need to be trained with the support of learning algorithms. The embedding of a system into a complex environment requires adaptive behaviour as one cannot foresee, during the development phase, all future situations the system has to react on. Although there exists no generally accepted definition of intelligence, there is general consensus that no intelligent system can exist without learning abilities and adaptive behaviour.

The reconstruction of the physical processes requires techniques for searching in a complex problem space. A good problem solving method has to be robust, i.e. goal-directed and systematic, and efficient. But, because of the combinatorial explosion, robust problem solving methods are most often not efficient. Search techniques can be subdivided into calculus based methods, enumeration schemes, random search, and random-directed search. Calculus based methods make strong restrictions to the search space. Practically, search spaces are most often uncontinuous, multi-modal and noisy. Therefore such methods turn out to be not robust. Enumeration schemes and random search are robust but not efficient. Surprisingly, random choice within a goal-directed search process (Chap. 3.5) leads to a very efficient search procedure [7].

After these general remarks we shall examine how symbolic and dynamic systems might be used for the construction of artificial “intelligent” systems.

3.2 Symbol Manipulation

The wide field of domains in which expert systems solve complex problems such as diagnosis, construction, planning, and scheduling shows the success of the symbolic knowledge representation paradigm. Symbolic knowledge representation has a clear advantage over other methods: the knowledge chunks are communicable, i.e. they can be explained, taught, learned and easily modified.

But, experiences in developing expert systems have shown insufficiencies partially based on fundamental deficiencies of the symbolic approach. So, most expert systems are not able to communicate with the user, contain no deeper knowledge about the problem
regularities, possess no common sense knowledge and don’t learn from experiences. Partial improvements are achieved with second generation expert systems by introducing user models, knowledge acquisition techniques, qualitative reasoning, or by storing a huge amount of knowledge. But especially the last point shows the limitations of symbolic processing, at least if one takes a human being as a measure for an intelligent system. The more knowledge a symbol processing system contains, the more time the system needs for the retrieval of knowledge. This contrasts with a human expert who accesses his knowledge faster, the more he knows. This so called “expert paradoxon” is probably only explicable with a highly parallel model. But different investigations showed that the capacity to integrate parallelism in symbol manipulation systems is very limited. Another point is that the symbolic approach lacks of inherent methods to refine raw incoming data into structured symbolic information and to deal with inconsistencies, such as erroneous data, missing values, and unexpected events. But both properties are required if expert systems have to be connected to complex external processes in order to solve tasks such as data analysis or control.

3.3 Dynamical Systems

The main difference between symbol manipulation and connectionism lies in the problem solving procedure. Solving a cognitive task using the symbolic approach of traditional AI means running through a chain of logical inferences while the problem solving process in neural networks is based on the dynamics and the ability of self-organization of such non-linear dynamical systems. In dynamical systems numerical variables in general evolve with time according to a finite set of equations. In complex applications, dynamical systems have a high degree of freedom and a big phase space with many different attractors. Often, natural dynamical systems have been transferred into a problem solving context: we find besides of artificial neural networks also random boolean networks, reaction diffusion models, evolution strategy, simulated annealing, immune system, genetic algorithms, and classifier systems. In neural networks we even find two coupled dynamics, one in the state space and another one in the weight space. The advantages of dynamical systems are robustness, fault tolerance, the ability to resolve ambiguities and contradictions, and the generalization ability. In most cases the dynamics is defined problem independent, i.e. such systems can be used also if only very little is known about a specific application. Disadvantages are the missing transparency, the reduced explanation ability, and the lacking stability of dynamical systems.

3.4 Connectionist Systems

Parallel processing, distributed knowledge representation, and adaptive behaviour are keywords of the connectionist approach. Connectionist models claim to capture more complex and imprecise properties of the real world. They are inherently parallel and furthermore adaptive in the sense that knowledge can be learned directly from experience. With respect to distributed knowledge representation, connectionist models offer quite new possibilities. In neural networks knowledge chunks may be represented either analog, or subsymbolic, or symbolic. In particular, neural networks offer the possibility of transferring analog representations to symbolic categories, or, of representing high-level categories from low-level data tuples.

Learning has a special importance in connectionist models. As in most cases, it is the only possibility to store knowledge in the network. The network processes training examples in a training phase and modifies the weights of connections according to a learning rule. An explicit registration of all the knowledge is not necessary. Instead, the autonomous development ability of the system can be used.

This advantage may prove at the same time to be a disadvantage. Especially when integrating higher level cognitive tasks, the problem of how to deal with previous knowledge appears [4]. One can partly overcome this problem in an indirect way during the phase of
connection engineering, i.e. the design of different layers and connections between several nodes, the selection of a proper learning rule, etc. Last but not least, neural nets have to be trained from examples and a net can only be at most as good as the examples offered. For human learning also the explicit exchange of knowledge and the existence of domain specific knowledge plays an important role. For the event reconstruction task we normally use several complex theories (invented by man) and it is not easy to imagine a neural net (trained only by examples) to be able to re-invent such theories.

We will bear in mind that connectionist models can cover a part of the spectrum of human intelligence. Especially for perceptive tasks they are very useful. But for the representation and manipulation of higher level knowledge connectionist models are less qualified.

3.5 Genetic Algorithms

Genetic algorithms [8] are an example of a problem solving method using random choice as a tool for guiding the search process. They are based on the evolution principle of nature, i.e. each species tries to adapt to a complex and changing environment in the best possible way. Only the fittest among a population of species survive and are allowed to reproduce. The features of each species are encoded in the “make-up” of its chromosomes. The chromosomal make-up can be altered by operators when parents reproduce. A selection is performed by prefering for reproduction chromosomes with above-average properties.

The dynamic of genetic algorithms originates from a sequence of selection and recombination steps applied to a population of chromosomes which code possible solutions. With respect to a possible implementation on a multiprocessor architecture, genetic algorithms have the advantage that they do not need a parallelization potential in the application itself. Instead, the problem solving method itself is inherently parallel. To speed up the search process, several alternative solutions are inspected in parallel and reasonably good part solutions are exchanged. The parallel procedure of the algorithm together with the parallelization ability of all steps within the algorithm leads to a highly efficient problem solving process on a multiprocessor machine [9, 10, 11].

Weak methods are problem solving methods that make only a few assumptions about the problem domain. Genetic algorithms belong to this class. Besides a function to evaluate possible solutions no further knowledge about the problem domain is necessary. However, if problem specific knowledge is available, one should try to incorporate it into the algorithm to get a more efficient method [12]. For example, local search methods could be used to build the initial population or to optimize the members of the final population. The classical operators (crossover, mutation etc.) performing a syntactical operation could be replaced by heuristic operators to realize a more clever recombination. These operators have to keep the property of implicit parallelism provided by the genetic selection rule. To do this they must be able to combine small building blocks to larger building blocks to reduce the dimensionality of the problem. If the available knowledge allows the construction of an approximate model of the problem domain, then this model can be used to compute the (perhaps erroneous) evaluation function. This makes sense if the computational cost of the evaluation function is high.

3.6 Hybrid Architectures

Symbolic and dynamic systems can be regarded as two counterpoles with respect to knowledge representation and processing. Although there exist applications where both techniques compete with each other [13], one can identify some substantial complements. Intelligent behaviour requires inferential (knowledge based, sequential) and dynamic (connectionist, parallel) mechanisms; it requires symbolic and non-symbolic representations. Two architectures are possible: A cooperative system where dynamic and inferential components work hand in hand on the solution of a complex task and a system where both
methods are combined in a new architecture, e.g. in a classifier system.

A cooperative system offers the highest degree of flexibility. At least theoretically, for each part problem the optimum problem solving method, the most suitable representation scheme, and the most efficient learning method can be chosen. Communication between the different subsystems can be performed by simple data exchange. Also a tight coupling is possible [14], or, as in Ref. [15], where a neural network communicates via spreading activation with a semantical network.

A tight coupling between dynamic and inferential parts can be found in classifier systems (see e.g. [7]). The idea is to integrate in a rule based system a dynamical part. Rules are typically used for inferences. But, together with association and evaluation, they (and with them the knowledge base) get subject to steady evolution and refinement. Three activity levels can be distinguished in classifier systems [16]. The kernel of a performance system consists of a production rule system which differs from a conventional rule based system by a subsymbolic rule representation and by parallel rule applications. The system communicates with the environment via detectors and effectors. A credit assignment algorithm valuates different rule applications and decides by this method the effectiveness of the different rules. In a discovery system the detection of new rules is based on genetic algorithms. The dynamics of the genetic algorithms depends on the evolution of a population of rules performed by selection and recombination steps.

All components of a classifier system are inherently parallel, i.e. one can reach nearly linear speed-up in CPU time on massive parallel architectures [9]. In classifier systems we find both concepts: the concept of self organization, parallel processing, and distributed knowledge representation known from the connectionist approach, as well as the advantages of rule based systems. It is possible to introduce previous knowledge in the set of initial rules; rule chains can be used for an explanation component.

4 A Hybrid Expert System for Event Analysis

As mentioned before, the whole task of event analysis consists of two parts which need different methods. A connectionist subsystem could be used to pre-process the data delivered by the detectors. Problems like feature recognition, pattern completion, and classification are ideally suited for neural networks. Work on this part problem has been started by several groups, e.g. [17, 18, 19]. Monte Carlo simulations deliver training examples for the learning phase of the nets. It should be possible with a chain of different neural networks to condense the quantitatively large amount of data with little information content in a single data word into a small quantity of higher level information. Such pre-processed data could then be the input to a symbolic working system able to reconstruct the physical processes which led to these data.

For the process of physics analysis we have chosen a system based on genetic algorithms. The concept of genetic algorithms had to be extended in many respects. While classifier systems are rule based systems with a genetic component, we went the opposite way and used a genetic algorithm with an inferential working component.

A chromosome in our application codes a possible physical state after the $e^+e^-$ annihilation process together with information about the evolution from this state to the observable physical end state. In the context of the initially mentioned word problem a chromosome codes a possible combination of letters (genes or particles) and in addition an (incomplete) inference tree of the word. From the global solution (the complete inference tree) it is possible to reconstruct the whole event.

The choice of a bit string code for a chromosome as used in standard genetic algorithms is no longer adequate. A gene (building block of a chromosome) is now a structured object which either codes an observed particle (terminal symbol), or a reconstructed particle (non-terminal symbol), together with a reference to the physical processes which created it (substitution rules). On a massive parallel computer, the efficiency loss expected from the more complex coding is at least partly compensated by not modelling a
whole chromosome on a single processor but by using one processor for each gene which results in an additional parallelization potential on the level of genes.

The initial population for the genetic algorithm is built up from chromosomes with randomly ordered terminal symbols (stable particles). In a hill climbing step we try to apply substitution rules on neighboured genes (particles) within a chromosome. The hill climbing component is realized in a parallel production rule system. The time used for the matching phase depends on the number of particles within a chromosome but not on the number of rules. The matcher is applied in parallel to different chromosomes which results in an additional speed up. In constant time intervals a subset of initiated (ready to fire) rules is selected probabilistically and executed. The developed hill climbing module can be interpreted as an indeterministic parser which performs a highly parallel step into the depth of the problem space. It represents the inferential part of our problem solving method.

The dynamics of the genetic algorithm is essentially given by the selection mechanism and the recombination scheme. The pressure of selection results in a growing quality of the total population while the application of recombination operators on selected individuals leads to the evolution of the population. The crossover operator is of particular importance. This operator is responsible for the exchange of genetic material of different chromosomes to a common offspring. The quality of single individuals is increased by the exchange of above-average part solutions (building blocks). Applying the mutation operator (random change of genetic material) enables us to proceed in up to now inaccessible ranges of the search space.

The application of structured chromosomes requires the design of complex operators. The crossover operator has to be able to inherit parts of the parent’s inference trees to a new successor chromosome. In addition we need the possibility to break up and withdraw inferences. The parallel application of those operators on any set of chromosomes is performed also in constant time intervals.

Concerning the implementation of the analysis task on the CM2, several problems had to be attacked. We could not deal with simple and disjointed data with fixed neighbourhood relations. But, we had to handle complex structured data tuples which were manifold correlated with each other. For the different components of the genetic algorithm, evaluation, selection, crossover, and hill climbing, data parallel algorithms had to be developed. For efficiency reasons, the expense of communication between processors had to be minimized.

For a first prototype system, a reduced physical model (only non-flavour changing decays) has been applied to reconstruct the decays of unstable hadrons into stable observable particles. The model has been described by a context free grammar with about 1000 substitution rules. First very promising results could be achieved. Especially the run time conditions of the several components could be validated.

It was and still is our goal to develop a robust and efficient problem solving method on the CM2 architecture. Selection, crossover, and hill climbing together realize an intelligent, highly parallel, and indeterministic backtracking procedure. With incomplete or noisy input data, the system not simply stops but is able to produce acceptable results. In the best case, such a system could be used as an instrument to discriminate between several physical models by applying an appropriate grammar. Detecting for instance several events without a satisfactory inference tree can be a criterion to reject such a model.

5 Conclusions
We have identified the physical event analysis task as a member of a class of problems which need systems which display intelligent behaviour on different cognitive levels. Such systems can be maintained by a combination of symbolic and connectionist mechanisms. In a cooperative architecture, a connectionist component could be used to couple a symbol processing system to an environment, i.e. in our case to the experiment. Both techniques
are so far developed that a loose coupling especially for our application should not be too problematic.

A homogeneous integrated architecture is found in classifier systems, which combine the two cognitive levels. But, classifier systems up to now did not prove in bigger applications, although there are in principle no limitations. The main disadvantage of today's classifier systems is based on the poor representation scheme, the limited match algorithm, the use of syntactical operators of the genetic algorithm, and the strong dependence on the parameter settings.

We have chosen the way to extend genetic algorithms by inferential components which should be successful for applications which are up to now the domain of standard symbol manipulation systems. The growing number of multiprocessor machines together with the parallelization potential of genetic algorithms increase the attractiveness of this method.

* * *

REFERENCES


Parallel Process Interfaces to Knowledge Systems, in Eckmiller, R., Hartmann, G., Hauske, G. (Ed.): Parallel Processing in Neural Systems and Computers, Elsevier Publ., North Holland, (1990);
A.B. Cremer, K.-H. Becks, W. Burgard, A. Hemker


[12] J.J. Grefenstette


Neural Networks for Triggering, Fermilab CONF 90/20, (1990).

[18] C. Peterson

[19] M. Gyulassy, M. Harlander
High Energy Physics Experiment Triggers and the Trustworthiness of Software

Thomas Nash
Fermi National Accelerator Laboratory
Batavia, IL 60510 USA

I. Introduction

For all the time and frustration that high energy physicists expend interacting with computers, it is surprising that more attention is not paid to the critical role computers play in the science. With large, expensive colliding beam experiments now dependent on complex programs working at startup, questions of reliability -- the trustworthiness of software -- need to be addressed. This issue is most acute in triggers, used to select data to record -- and data to discard -- in the real time environment of an experiment. High level triggers are built on codes that now exceed 2 million source lines -- and for the first time experiments are truly dependent on them. This dependency will increase at the accelerators planned for the new millennium (SSC and LHC), where cost and other pressures will reduce tolerance for first run problems, and the high luminosities will make this on-line data selection essential. A sense of this incipient crisis motivated the unusual juxtaposition of topics in these lectures.

II. What's the Big Deal About Software?

Computers are fundamentally different from other technology used by society in two basic ways: the degree of system complexity and the level of human involvement. Measured in terms of the numbers and dynamics of their internal states, computers are systems of extreme complexity. This complexity carries some of the apparent opaqueness and unpredictability that is characteristic of the human personality. The sometime resemblance to human capabilities is what research in artificial intelligence (AI) tries to exploit. Computer technology is the highest of high tech. Yet, unlike other technology, it is extremely difficult to quantify. What do you mean by complexity when you don't know all the system states? What do you mean by reliability when you don't know all (and, probably, not even most) of the failure modes?

Also unique is the degree of intimate human involvement with computer systems. The human component is obvious and essential for specification, programming, definition of input, interpretation of output, and attempts to confound the system. Humans are involved in some of these activities in other enterprises, such as the development of experiment detectors, bridges, and vehicles, but never with anywhere
near the intensity required for computers. What is different is that computers aid the intellectual rather than mechanical functions of humans. These are attempts to replace our most complex and "human" capabilities. Their specification and design require a deep and most intrusive self-analysis of what we are trying to accomplish. The intensive human involvement is a profound and often neglected aspect of the computer problem.

Computer systems must be considered as part of the overall system in which they are embedded, not as one "of a collection of components" or as "appliqué". In particular, it is essential that they be understood as human-computer systems. When something goes wrong, it is a failure in "integrating a complex, highly contingent world of analog variations with both human subjectivity and discrete-state machines."

Computer hardware is subject in many, but not all, ways to the standard techniques of reliability engineering. Redundancy has been exploited to produce extremely reliable duplex processor machines for situations requiring high availability, such as banking, airline reservations, and telephone switching. Telephone switch downtime, planned and unplanned, is less than two minutes per year in practice. The requirement on the development, sponsored by the Federal Aviation Administration, of an automated flight-control system, including its emergency mode, is for less than three seconds down time per year. Most important, it is possible to measure hardware reliability in terms of mean time to failure (MTTF) and to predict MTTF from measurements on components for a system in which the overall complexity is kept under control. This process is what allows one to design for improved reliability with redundancy, component selection, and reduction of component count.

In computer software the situation is entirely different. Hardware performance and reliability have improved in the last 25 years by factors like a million and a thousand, respectively. There has been barely an order of magnitude improvement in any performance or reliability aspect of software – despite extensive research directed at what is ominously referred to as The Software Problem. Why is this the case? We use computers because they handle massive amounts of data that they can manipulate with data-dependent paths through complex logic. Convoluting the data states with program paths leads to a huge number of internal and final states. No human being can specify all the states for any but the most restricted applications. No human programmer can conceptualize all the states. No human tester can test all the states.

Why can’t we test software until it can be certified bug-free? Because, as E. W. Dijkstra said, “Testing can show the presence of bugs, never their absence.” At the beginning of his book on software testing, Glenford Myers provides a simple homework assignment for his readers. The idea is to point out how difficult it is to test fully software written to even the simplest of specifications. The spec: read 3 integer values which are the lengths of a triangle’s sides; print a message stating whether the triangle is scalene, isosceles, or equilateral. What test cases would you define for a program written to this spec? Experience with real bugs for this example suggest that the test cases must satisfy at least 14 issues. (Did you check the case (0,0,0)?) A group of highly experienced programmers averaged only 7.8 out of 14. This problem is so simple that formal logic
methods could easily be used to prove the program carries out its specified task. The code required is a few tens of source lines. For large problems involving several million source lines of code (megaSLOCs), formal methods are impossible, and even highly experienced programmers would be able to define tests that would catch only the tiniest fraction of possible bugs.

Everything said here about software applies to firmware. Firmware involves data, program instructions, or logic that is fixed in the memory or the wiring layer of an integrated circuit. The name firmware applies to read-only memory (ROM), programmable logic arrays (PLAs), gate arrays, and a large variety of other devices complex enough to contain extensive information. Generally, this information is prepared in a variant software language. What goes into one chip may consist of thousands of lines of code. The problem of firmware may be even more insidious than the problem of conventional software. Firmware and hierarchical design languages (HDLs) are so intensively used in designing digital systems like computers that increasingly one sees hardware systems whose functionality traces almost entirely to software programs burned into silicon.

Redundancy works for computer hardware and for non computer systems; why not use it for software? The problem is what do you make redundant? Running two identical copies of software (on reliable hardware) will give the same result, right or wrong. The failures in software are those of the humans in this aspect of the human-computer system. So what about redundant human programmers? The idea is referred to as n-version software development.7 Several independent and isolated teams build software to the same specifications. The results are compared. If the number of versions, n, is greater than 2, there can be a vote. The software can even be run in separate computers. Multiple-version software and multiple, isolated, independent testers are used in critical situations to improve reliability, despite the extra expense. One of the five computers on each of the space shuttles runs a second version of the software. (In fact, a delay in the first shuttle launch was caused by a veto from this computer. The veto resulted from a synchronization bug that had 1 chance in 60 of occurring.)

Without question, this technique improves reliability, but much less than one would expect from reliability theory for systems other than software. In hardware, if the probability of failure of one component is p, the probability that a system of n independent (parallel) components will fail is p^n. If p is small, even n = 2 will result in very high reliability. However, the redundant components in software are human programmers or testers, and they are not independent. Human errors are correlated by culture (humans read the same textbooks and learn the same algorithms) and by genetics (the thought processes in different brains may differ, but not all that much). n-version techniques are not a panacea that will eliminate software errors. As an aside, they can be useful in controlling hostile infiltration of software development groups, an issue hopefully not relevant in high energy physics, but nonetheless instructive. If there are n, independent, geographically isolated teams developing the same software, the probability of infiltrating all of them successfully does go as p^n.
Measuring software reliability is also a serious problem. Normally, to get a measure of MTTF, one averages the failure rate in a large sample. But the concept of multiple "samples" of software has no meaning. IBM has developed techniques that get around this problem for certain software products. Their answer is to measure a large sample of different usage. IBM measures the rate at which errors are found during an extensive debugging process involving release of the software to an ever-increasing circle of users. This technique is very effective for the errors of concern to IBM, which is "interested in failure-free execution intervals, rather than ... [an] estimate [of] errors remaining" IBM finds that 33% of errors are found after 5,000 user-years of being exercised (which IBM defines as 5,000-year MTTF) and therefore can be caught in a short time. The goal of these measurements is to determine a product-release date.6

What is special about the IBM experience is that it deals with compilers, linkers, operating systems, and other products that have a huge and diverse circulation. Users exercise all aspects of the software and quickly find the problems that most other users will find; these are the problems that matter to IBM's overall reputation. In other situations, the concern extends to the sum total of all the uncommon errors, those that could, for example, result in the crash of an airplane -- or a fatal systematic error in an experiment's trigger system. In fact, so-called fly-by-wire software can at best be measured with present techniques to have a MTTF of $10^{3-4}$ years. This compares to the normal standard for commercial aircraft components in the neighborhood of $10^6$ years.

III. Is HEP Software Different?

What is special about experiment software compared to the IBM experience is the small number of users from a specialized, highly computer literate community who are generally quick to learn the capabilities of software and accustomed to a short time frame when responding to problems. The small number of highly similar, very sophisticated usages of a program in high energy physics tends to test the farthest corners of program space at some point in a product lifetime, but generally not early in the product life as in the IBM case. Software projects are interconnected communal efforts with many individual perspectives, software styles, and methodologies, and with varying receptivity to discipline. Multiple institution management results in limited management controls over the communal project. Furthermore, in a surprising similarity to military software, critical testing of HEP software and its use are essentially simultaneous. The actual use determines the true requirements. Implementation must respond to changes in understanding on the battlefield of experiment data runs. Repeated use of the software takes place in an environment where users, technical concerns, and understanding of goals are steadily changing. Requirements must, therefore, evolve, and iterative development of software is natural, appropriate, and inevitable.

Computing is essential to HEP experiments at four key stages: triggers, data acquisition, reconstruction, and analysis. Detectors consist of often huge assemblies of
electronic detectors that signal the presence and time of passage of ionizing particles. In many cases detector pulse height or area also indicate the amount of charge left by a track. The detector analog information is quickly converted to digital form. There is generally too much data to record, and experiment triggers are used to select events of interest for off-line analysis. The process of assembling and calibrating raw data from different parts of the detector and recording it on permanent storage media is known as data acquisition. The data is normally assembled into “events” that correspond to a single primary particle interaction. The raw digitizer data must be transformed into physics variables (momentum, mass, and originating vertex of each particle). The pattern recognition and regression procedures are known as data reconstruction. The resulting data summary tapes (DSTs) contain, in principle, all that was measured about the physics variables of detected particles in typically billions of interactions. The DSTs are subjected to data analysis by physicist written software that searches and measures new physics using simple statistical visualization displays and sophisticated fits.

IV. Multiple Levels of Triggering

The triggering of high energy physics experiments proceeds in a succession of steps that each reduce the data that must be recorded. One can identify four typical levels of triggers in the more complex large hadron collider experiments. Figure 1 shows the present conceptual data flow and triggers plans of the Solenoid Detector Collaboration (SDC) in preparation for the first SSC experiment. This structure is also seen on a simplified scale in smaller experiments and those dealing with lower data rates at electron machines.

At Level 0, fast analog electronics are typically designed to identify anything that implies an interaction has occurred. This may be an OR of many detector element signals or an indication that there was a beam crossing or a tagged beam particle. This trigger takes tens of nanoseconds and is used to gate front end electronics whose incoming signals had been cable-delayed to cover the decision time. It also starts the Level 1 trigger.

Level 1 triggers use the earliest available subdetector data. This data may be still in analog form, or it may have been digitized by a fast analog to digital converter (FADC), or a mixture of both. The trigger might require, for example, a significant amount of $E_t$ or a $\mu$ track. This is intended to reduce the rate into the Level 2 trigger. In fact, Level 1 triggers are often integrated as the first step of a Level 2 trigger. These are very special purpose processors, often implemented in ECL, and generally programmed by changing parameters in a DRAM or firmware. Their time scale is about 1 - 2 $\mu$sec.

The Level 2 trigger occurs on a time scale of 20 - 50 $\mu$sec. It decides whether to permit a full digitization and readout of all detector signals, a time-consuming and dedicated activity that, if triggered too often, will increase the dead time of the experiment. This trigger level uses as much subdetector data as is available from FADCs and builds on Level 1 information. It often includes calorimeter cluster finding and some track
finding. Typically a processor which can be (wire or micro) programmed is used here. In the past, DSP and bit slice technology, as well as data driven processors, were commonly applied. These are generally not high level language programmable devices. The sophisticated techniques required for programming them limits access to a very small group of skilled people, effectively hiding and protecting the details from most of the members of a large experiment collaboration. Collaborations commonly interact through a trigger language or table that permits making complicated trigger choices without having to meddle with the detailed software.

At Level 3 all data has been digitized and is available, making possible triggering on a global reconstruction of data. This trigger permits recording on tape and takes 100s or 1000s of msec per event to decide. Parallel farms assembled from now generally off the shelf RISC workstation technology computers, and programmed with MSLOCs of FORTRAN, are used in an environment that is compatible with the off-line computing of the experiment. Most of the software may be almost indistinguishable from off-line. In fact, successful large program Level 3 triggers to date have depended on off-line reconstruction codes that had been thoroughly seasoned by use on a previous run's data.

The perceived availability of so much "free" computing results in Level 3 processing often being applied to data formatting and event building. This is the only real difference from off-line reconstruction and is a sign of a new trust in these systems by collaborations. Previously, specialized event builder processors were used to assemble data from across the detector that corresponded to a single interaction event, and then to pass it to the Level 3 trigger. The D0 experiment at Fermilab established a new direction by sending subdetector data on multiple data cables directly to dual port memories in their highest level trigger. A switch approach like this will also be used at the Collider Detector at Fermilab (CDF) and the SDC. CDF is using a commercial network switch (Ultranet), further demonstrating the trend to off the shelf hardware at the highest levels of data acquisition and trigger systems. Although D0 is planning to attach co-processors (Level 2-like special processors) to its commercially available micro-computers as accelerators, collaborations have clearly moved toward commercial systems that have a setting as similar as possible to the off-line.

There remains one big difference with the off-line, and this is the matter of trustworthiness in a context where data is being discarded without means of recovery. This is a critical subject to which we will return after a brief digression on the place of special purpose hardware in the on-line HEP scheme of things.

V. Special Purpose Hardware

Triggers have traditionally been seen as an opportunity for advanced and creative application of special purpose computers. Bit slice, data driven, systolic, associative string, and digital signal processors are all technologies that have been applied, or at least evangelized as appropriate.9 None are off-line compatible. They all require large
amounts of microcode, hardwired programming or other talent intensive effort. Ostensibly they have been used because of speed and timing requirements. Perhaps, they have also been a way to hide critical processes from prying hands behind a wall of esoteric technology. Even at Level 2, it is now possible to use high level programmable processors efficiently in a data gathering architecture. An example is the transputer trigger/DAQ system being used for ZEUS at HERA in Germany. Future switch based parallel processor standards such as the Scalable Coherent Interface (SCI, IEEE 1596) will make it straightforward to use commercial high performance, high level language programmable processors such as RISC in these applications.

So why use special purpose processors? Are they faster than RISC? A special purpose processor clocking at 50 MHz with one instruction per clock is not significantly more efficient than a streamlined RISC processor executing one instruction per 50 MHz clock. The component economics of digital processor architectures, esoteric or not, are basically the same: the number of transistors required for an instruction operating on a unit of data is almost a constant of nature. The speedup potential of incompatible special processors no longer seem compelling. Their use as subroutine coprocessors now seems even less compelling since here Amdahl's law applies: if $f$ is the fraction of time spent in code to be run in the coprocessor, the speedup for an infinitely fast coprocessor is only $s = \frac{1}{1-f}$. If $f$ is as high as .75, $s < 4$. In a trigger, events are discarded as you go so one can focus the speedup on early stages, but as we just noted, there isn't much, if any, speedup available from special hardware.

The exception to this poor speedup prognosis for special purpose technology is when one can train the electronics in advance and look up answers in real time. Neural networks and memory look-ups (MLUs) are two approaches that permit this. Neural networks are analog networks with weights that are allowed to relax to a state corresponding to an answer that depends on an input data pattern. They are trained by setting weights in advance by feedback (in the HEP context) with real or Monte Carlo data. Their strength is pattern recognition and they are being applied to triggers and reconstruction where it is desired to identify B physics events or separate gluon from quark jets. MLUs are large memory tables which are trained by loading results (computed in large FORTRAN programs) for all inputs (addresses). To look up an answer, one addresses the memory with the input variable and the table delivers the stored computed function of the variable as output. MLUs are particularly appropriate for quickly computing complicated functions.

Neural networks were covered extensively by other lecturers at this summer school. For MLUs the speedup potential is (almost) unlimited since in principle you can compute forever to load them up. Memory technologies have improved so much since the introduction of MLUs\textsuperscript{10} that their practicality has crossed a new usefulness threshold at 32 bits. A single bit MLU function (to trigger or not to trigger) of a 32 bit variable in 100 nsec is now possible for about $35,000$ with 135 SIMMs fitting on a Fastbus sized board. By mid decade this should be down to $2000$, allowing a full 32 bit function of a 32 bit variable for $65,000$ on 8 VME cards. Loading these big MLUs from a
disk image will take about a minute per output bit at 10 MBytes/sec. 16 Gbytes of disk will be required for the full 32 bit function. The practical maximum time that one can imagine for computing such a disk image is a 100 processor farm for a day. Assuming mid-decade processors, and an unoptimized loop, the corresponding speedup is at least $2 \times 10^4$. So maybe MLUs have a future again. Since FORTRAN programs can be used to program them, MLUs are unique among special purpose systems in that they can be programmed in an environment compatible with off-line software.

VI. Trustworthiness and High Energy Physics Software

The trustworthiness of something, to paraphrase the excellent definition by David Parnas, is our evaluation of the probability that it will not cause something terrible to happen. What factors encourage high energy physicists, under the difficult circumstances of their experiments, to trust software dependent systems? How are subjective perceptions of trustworthiness developed? We will approach this business like question in a business school manner, by looking at some case studies, in the context of triggers at Fermilab's collider experiments, and drawing a set of conclusions, somewhat tongue-in-cheek and with no intent to offend.

In the past, although many experiments boasted high level triggers, few depended on them. The upcoming collider run represents the first time that major hadron collider experiments consider Level 3 farms to be a necessity. At the anticipated luminosities CDF needs a full factor of 10 reduction in its Level 3 compared to ~2 last run, and D0 is counting on a factor of 200. So, our first possible conclusion: Dependence breeds confidence?

CDF's Level 3 was used in several previous runs. After serious difficulties in porting 2 MSLOCs of rather unstructured off-line code to the inadequate environment of the first generation of Advanced Computer Program (ACP) MC68020 based farms, the system ultimately worked and was trusted to a ~2x cut. Once it stabilized, the Level 3 hardware and software worked well enough so that it made only a small contribution to the experiment's logged down time in the 1989 run. "This may explain the experiment's trust now," a physicist responsible in this area said. "Everyone is asking to put more into Level 3." For the 1992 run, the off-line software is now mature and trusted off-line modules will be used in the trigger. The new Level 3 trigger will be based on a SGI Powerserver RISC farm, which has a development environment that will also be used for off-line Unix reconstruction farm processing. So a second possible conclusion: Familiarity breeds confidence?

At D0 the equivalent of what we have defined as a Level 3 trigger is known as "level 2". It is a farm of DEC microVAX computers which corresponds to the dominant computer flavor used by this collaboration for its on and off-line activities. D0 is pushing very hard to catch up with CDF in this coming run and cannot afford an "engineering run". Therefore, they do not have the luxury of mature, run-tested off-line software to apply to their trigger. A set of special filter programs is being developed.
There is a push by those responsible to establish a “Filter Certification Board” in order to convince the collaboration that the high-level trigger is ready. This board would formally review the independent physics testing of triggers already in place. This is to preempt any temptation to use the level 3 only loosely since some might argue that a Top discovery could most safely be obtained in that way. Computer upgrades and the coprocessors are arriving late enough in order to make a few collaborators nervous, though in fact there should be an adequate 6 months to stabilize this hardware. So, two more possible conclusions from this case study: *Time pressure breeds confidence? Hardware reliability influences trust in software?*

CDF’s Level 1 - 2 triggers are another interesting case. They are highly trusted by the collaboration and have been for some time. The parameters for gain calibration are entered via digital tables and DACs. The collaboration incorporates its trigger decisions in a “trigger language” which is very readable and is, in principle, verifiable by careful checking. Key functions are structured into coded routines called by the trigger language. The hardwired and microcoding is the restricted domain of a few experts, which automatically puts it under tight change control and results in its being highly structured, well defined, and testable. Our possible conclusion here: *Esoteria limits access (to experts) and forces structure and project discipline far more than in open high level programming environments?*

In some ways D0 low level triggers (referred to as level 0 and 1) are similar, but because of the intrinsic simplicity of their liquid argon detector which directly measures charge, calibration is easier. Charge is injected to calibrate amplifiers and the gain, and trigonometry, is hardwired into fixed resistors which are selected and installed subject to standard, careful, quality assurance. There is some flexibility for corrections and other purposes in PROMs which are, of course, programmable. The use of firmware is very effective change control. “Burn ‘em and forget ‘em,” was the philosophy expressed. Just as at CDF the low level trigger functions are the responsibility of a small skilled group. The collaboration’s trigger choices, overseen by a Trigger Board, are incorporated in “COOR” a coordinating on line process. So our last two possible conclusions: *Geometric and calibration parameters are subject to routine QA? Reduced complexity increases trustworthiness?*

I chose to use the two Fermilab collider experiments as study subjects for obvious reasons of convenience in gathering specifics. However, it should be clear that the “possible conclusions” we identified in these case studies about how HEP experimenters establish trustworthiness would be the same with CERN experiments as examples, or experiments from any other HEP laboratory. The way in which we establish trustworthiness in our experiments is subjective and depends on an intensive and careful study of the consistency of data. Compared to what goes on outside HEP, this bottom line approach actually looks pretty good. The question we turn to now is whether it is good enough.
VII. A Crisis in HEP Software? How Do You Measure Complexity?

Some of us who inhabit the no-man’s land between high energy physics and computer science, reacting probably more on aesthetic grounds than anything else to the state of HEP computing, have made hysterical sounding statements about a “crisis in HEP software”. Is there a crisis?

Yes: Our detectors and collaborations are getting bigger and more complex.

No: We are doing sort of OK, intuitively structuring our problems. And it ain’t broke now, so let’s not risk fixing it.

I think I agree that things are not all that bad right now, but to understand what might change for the next generation of experiments, we need to understand a few things about complexity and about how we manage the software in HEP which deals with this complexity.

The first question is how do you measure complexity? The computer software industry makes a big effort to measure complexity. Accurate cost estimation is a critical factor in the profit equation, and it depends on an early understanding of complexity in a project. If you estimate a job wrong and underbid, it can be very costly. Extensive complexity estimating methodologies exist,\textsuperscript{14} based on approaches like Function Point Analysis.\textsuperscript{15} Such techniques look in great detail at the specifications for a job, accounting for what has to be done, module by module, and making reference to extensive experience-based tables -- almost like an actuary defining risk for insurance purposes. The methodologies and the experience are aimed at large software projects under tight management control. No work has been done on understanding how to estimate costs in large research-oriented software projects. A study in this direction would seem to be motivated by the number of pending big scientific projects, anointed and unanointed “grand-challenges”, that depend on major software efforts.

In the absence of appropriate complexity-estimating techniques, we can take a physicists style look at big-picture quantities: the number of detector types requiring separate software efforts (we really should count “regions” -- CDF has ~100) and the number of collaborators (individuals/institutions). The detector channel count does not add to software complexity although it, of course, does affect production computing resource requirements. Here are estimates for three US hadron collider experiments:

<table>
<thead>
<tr>
<th>detector types</th>
<th>collaborating individuals</th>
<th>collaborating institutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF</td>
<td>15</td>
<td>332</td>
</tr>
<tr>
<td>D0</td>
<td>9</td>
<td>307</td>
</tr>
<tr>
<td>SDC</td>
<td>9+</td>
<td>~700+</td>
</tr>
</tbody>
</table>
This suggests that detector complexity is not qualitatively changing. If all that matters is detector complexity, software may not get much worse at SSC/LHC. However, a detailed scrutiny of the impact of individual subdetector complexity on software has not been done here -- nor has it been done by collaborations, and it is not being done for SDC. This level of complexity may be the straw that breaks the camel's back. A comparison of the CDF startup experience with D0 will be instructive, since it has been argued that D0 is a less complex more homogeneous detector.

A part of the problem is known to scale with the number of software workers. The individual institution (university) group size seems to stay constant. Software team sizes are unlikely to change because of small group psychology issues. By a Parkinson's law kind of argument, we can expect that the resulting increase in number of software teams will result in more different things being done (more physics, more triggers, etc.) Each team will continue to interact with about the same number of teams so the total interaction complexity would seem to scale linearly with the number of people. That in itself is not a troublesome conclusion. However, Stu Loken at LBL points out that there will be an increase in the number of groups interacting with key core groups -- those with the biggest responsibilities. The complexity of key subsystems, like trigger supervisors and production managers, will be increased and their reliability may be impacted.

VIII. A Crisis in HEP Software? How is HEP Software Managed?

The other question we need to address in evaluating the extent of any future software crisis is how, in fact, is HEP software managed? First of all there is an intuitive structuring of activities. Subdetector modularity is carried into software, with hardware developers often doing "their own" software. The physics modularity is carried into software. Physics topics groups generally are responsible for overseeing topical modules of trigger and analysis software. Also as we have seen, the most critical triggers are typically handled in hardware and microcode which forces structuring and information hiding in an apparently natural and painless way -- at a considerable cost in talent.

Skeleton packages are developed by core groups and are used both for on-line trigger software and for off-line reconstruction and analysis. Physicist programs are plugged into these skeletons which provide data banks, manage batch production or trigger, sequence software subsystems, and move data from and back to tape. We will discuss later the software engineering concepts of information hiding and objects. Information hiding, which could go a long way toward improving productivity in these situations, does not appear to be applied significantly yet. The struggle to develop and maintain these packages is painful and difficulties in converting from one platform to another are symptomatic of problems that information hiding cures.

Master librarians have the role on experiments of approving inclusion of software for production and triggers. Some are assisted by review boards. The system works well if
the individual is strong and has the charisma and/or authority to impose some discipline. It is hard to identify such qualities in advance.

Perhaps the most distinguishing feature of the HEP style in software is that the fundamental approach to testing is through physics results. The requirement is: get the physics right. Most physicists do not know the meaning of “requirement” in the context of software or projects, and there is no other requirements document. The only accepted critical test of software is to look at results with real data: to see if trigger rates are reasonable, if event distributions are reasonable, and if anything at all looks fishy. Before there is real data, similar kind of testing is effectively done with Monte Carlo and cosmic ray data. However there is no systematic software testing, as that is commonly understood, at the component, subsystem, or system level. Furthermore, the testing tends to be schizophrenic self-testing. Independent testing by separate testers is rare. Independent testing from requirements documents does not exist. That concept is foreign to HEP.

Finally and rather unfortunately, the impact of detector complexity on software is still not a design consideration for the next generation detectors. The operative mind set, "Software has always been free. Why should we pay for it now?" was expressed at a senior level on one experiment. The real question is: is it free? And if it were cheaper, could physicists do more physics?

IX. A Crisis in HEP Software? What Has Changed.

If we look to the future, one can only anticipate that pressures will increase for large detectors to work reliably on their first time up. The history at LEP and CDF was similar: the biggest problems were in the systems with highest complexity, data acquisition, triggers, and on-line software. All of the experiments had sufficient off-line software so they were able to address "straightforward" analysis (W, Z) quickly (using an "express line" analysis at CDF) and get key first run results out. More complex areas (like B physics at CDF) were deferred to later. None of these experiments depended on high level triggers in their early running. CDF will be dependent on Level 3 this coming run, but now they have the confidence allowed by relatively stable and mature off-line software. At D0, as we have noted, it appears that the high level trigger may be considered critical in the first run. The D0 experience will be important in projecting toward SSC/LHC. SDC will likely depend on a high level trigger in their first run. It is not clear whether there will be available a straightforward analysis opportunity (like W and Z) for the first run that will promise definite results. The cost levels are high, and so will be the resulting visibility from the public, congress, and government oversight. Results will be expected quickly and one cannot anticipate much tolerance for first-run problems.

To summarize the present situation with HEP software, it is clear that how we deal with software is, at the least, unesthetic to cognoscenti, and that the issues are difficult to identify and verbalize. Although the complexity of whole detectors is increasing only
moderately, the impact of subdetector complexity (the region count as Bob Kephart puts it) on software is not yet a strongly recognized hardware design consideration. Collaboration sociological complexity will increase somewhat, but the impact will be felt most by critical core software groups. Collaboration trust is based on indirect criteria. Approaches to software quality/trustworthiness management are currently based on very talent intensive activities which weigh hard on physicists, particularly those who might otherwise be applying their talents to more physics productivity. None of this is really new.

What has changed qualitatively, it appears, is the dependence on high level triggers. Unlike the off-line, when a trigger fails you cannot rerun data tapes into it. Data, time, and money have been lost. With the new budget scale of experiments, equipment, and operations approaching a billion dollars, we cannot afford first-run failures. The field of high energy physics will be more dependent on a few experiments. The expectations are too high and the experiments too critical to the survival of the science. All this will be amplified by the increased level of governmental oversight which expects more professional management. It is hard to believe, looking at the record in the defense sector, that this oversight will not, sometime soon, be extended to software, which will correctly be seen as critical to the success of the taxpayers' huge investment. These are all changes that require a new scale of trust in software. This is exactly in the spirit of Parnas' definition of trustworthiness: we have to reduce the probability for software inducing something terrible to happen.

X. Traditional Approaches to Engineering the "Software Problem"

The "software problem" perhaps may be understood best in terms of software's not being amenable to the usual kinds of quantitative-design disciplines practiced by engineers dealing with other technologies. The problem persists in spite of being recognized and attacked. Since as early as 1960, software engineering\(^{17}\) has addressed the issues of reliability in human programming of computers. In 1972 Frederick Brooks wrote the classic book on this subject based on his experience managing IBM's OS/360 development.\(^{18}\) It is amazing how little has been added by others in the years since.

Traditionally, efforts have focused on means of organizing programs and making them understandable and readable. During the 1960s IBM provided free-flow charting templates to programmers. It is still considered undisciplined to fail to use such tools (or modern equivalents) to prepare a diagram of the logic flow of a program prior to starting on it. The flow chart should show what happens to data, from when it is input to when it is written out, including conditional actions and branches in the program that depend on the immediate state of the data.

In the 1970s a number of different methodologies evolved and were promulgated. These formed frameworks intended to define paths toward analyzing requirements and then writing programs that are well structured and, thereby, more readable and understandable, less error-prone, and easier to test and debug. Now often generically
lumped under the terminology *structured analysis, structured design* (SASD), these methodologies are much used. In the 1980s they have been incorporated into computer-workstation software as computer-aided software engineering (CASE) tools. CASE systems are now marketed by over a hundred, mostly start-up, companies. The crusade for more structure in software has left its impact on programming languages with "structured" constructs like do while and if then. Programmers have been steered strongly away from from using conditional and absolute go to ... branches, which, according to the dogma, represent the evil that results in much unstructured, disorganized code. SASD and CASE tools have been valuable and have improved productivity, especially for complex programs, but they have not proven to be a panacea as some had hoped. The fact is not surprising if viewed in the historical context. One experienced software engineer comments that SASD and CASE are "just glorified flow charting, no different from the 60s."

CASE tools have been used on high energy physics experiments with some limited success. These have included SASD tools as well as a HEP developed entity relation system called ADAMO which takes data entities from the screen down into the data structures of the code. The problem has been with the lack of a complete integrated CASE package. Paolo Palazzi, the developer of ADAMO, commented "... most toolkits abandon you after design. ADAMO stays with you ... but supports only one kind of abstraction..." For real acceptance in the research community, these tools must become more complete and manifestly productivity enhancing.

Another traditional feature of research focus for software engineers has been in the area of large project management. Projects are divided up into phases of requirement, specification, design, coding, and testing, very much as in other fields of technology.\(^{19}\) Diverse measurements support the common belief that the cost of errors increases by something like an order of magnitude for each project phase that passes before they are corrected.\(^{20}\) Extensive project-management methodologies are in place at IBM, AT&T, and other major software and computer-system developers. These insure that there is a careful review and documentation at each phase in an attempt to catch problems early. Design and specification changes are carefully tracked and controlled because of their possible impact on other parts of a system. When the emphasis is on the quality of reviews, rather than on paperwork and bureaucracy, such project management is very effective, perhaps the most effective means to ensure production of reliable software that does what is wanted.

The Department of Defense has recognized the importance of such project management. It promulgates software standards like DoD 2167A which defines a rigorous and complex sequence of reviews and documents for DoD software contractors.\(^{21}\) DoD 2167A requirements are so pervasive in the area of defense software that companies with CASE tools for the aerospace sector advertise "automatic 2167A document generation." Contractors staff offices just to produce the requisite 2167A paperwork. A defense contractor executive with extensive project experience put it this way: "Contractors set up two teams, designers and book writers, to meet the requirements [of 2167A]. It is coincidental if the results of the two teams are the
same." Military officers (like physicists) do not pay enough attention to the very difficult requirements phase, and give-and-take between those defining requirements and those specifying and designing a system is rarely adequate.

Although 2167A allows changes, it is normally used in one pass from requirements through design. The process is very difficult and results in requirements that bear little resemblance to reality. Requirements are typically 5 to 10 pages long per thousand lines of code, though there are many cases well outside this range in both directions. They are generally neither read nor reviewed properly, certainly not when the code "is over a million lines in scope". The project can’t be stopped if, at any stage of the review sequence, it proves unsatisfactory. The executive continued: "Economic forces say proceed. So pretty soon you are reviewing something different. The books are irrelevant. You eventually have to go through a very painful merge because of the final tests." The result is a bottom up, rather than an intended top down design. The 2167A "process fosters problems by being too detailed." The result is that irrelevant issues get in the way of understanding what the project really needs to be able to accomplish.

XI. Modern Approaches to Software Engineering

CASE tools and the way in which DoD 2167A or other rigorous review methodologies are conventionally applied represent the traditional approach to software engineering. More than this is clearly needed. Recognition has grown widely in the last decade of two concepts that are key to further improvement in software reliability: evolution of requirements and software reuse. A realistic and complete understanding of requirements is central to the goal of software that is reliable, effective, timely, and economically affordable. Big, complex projects are just too difficult to define completely from the beginning. In the research situation, successful evolution of requirements can be accomplished by keeping documents of requirements short at first with general descriptions of what is desired. As a design proceeds, the client and designers must work together on evolving and customizing requirements. Such a process requires a commitment of time and competent personnel to a continuing requirements process. Nothing is more important for obtaining software that has a chance of being truly trustworthy during a run.

The difficulty is focusing the attention of clients, scientific or otherwise, on the requirements process. This problem is exacerbated by frequent rotation of personnel. Those with the initiative to start a project are often not around to see it through to completion and thus to influence a consistent evolution of requirements. Nonetheless, clients are rightly most concerned about their interaction with the computer system. This point of interaction is the human interface, where users input requests and receive output as display screens or paper reports. The technique of rapid prototyping is proving very effective in allowing quick turnaround of ideas in the development of human interfaces. Special or general-purpose computers are set up so that displays are easily changed. No consideration about performance or computer cycles is expended during prototyping.
The process of rapid prototyping involves clients deeply in the decision-making process for requirements. A quick creative process to determine what a system will be like, using (often glitzy) color displays, seems to attract and hold clients’ attention at this critical phase much better than the traditional exchange of arcane documents. This fun approach to requirements focuses on displays and screens, exactly where one should focus to get started on a proper evolutionary top-down design. Unfortunately, the benefits of rapid prototyping and extensive end user involvement, are less applicable in some areas, such as in the guts of reconstruction and trigger decision programs, where there is little need for interfaces to humans during operation.

The deep involvement of users in the requirements and design process is so important to the ultimate success of a system that creative management methods are being applied to encourage such involvement. The problem comes in breaking away key client personnel from their ongoing responsibilities, the very responsibilities that the new computer system is supposed to assist. In a very successful and interesting experiment, DuPont isolated teams of users and software developers for periods of as long as six months. Each team was given a basic project definition and a “time box” in which it was to complete the new software project. The teams were to be rewarded individually and together and celebrated in the house newsletter, if they succeeded within the allotted time. All teams in a trial succeeded, and the savings in cost to the corporation, compared to estimates based on experience with traditional approaches, was extraordinary. On average, the seven initial projects came in at 33 percent of the estimates. The key to this strategy was the use of rapid prototyping techniques, combined with full-time access by software developers to knowledgeable and competent users. The user-clients’ understanding of their own requirements, and the realities of implementing them, deepened during the development process. In this way, not only were systems completed with great efficiency in the use of personnel, but the systems were also more likely to meet the real needs of the organization.23

Rapid prototyping can be made more efficient by software storming, a short and intensive software development effort intended to get a first order approximation of the system requirements. According to an individual familiar with this approach, “Practice has shown that the issues uncovered in this [“storm’] are the issues which require the most attention throughout the prototyping period.”24 Prototyping is, in some sense, testing in advance. Therefore, just like testing, prototyping is subject to missing out where scenarios are not tried. Here one never finds the last requirement, as in testing one never finds the last bug.25 Involving end users in the process is not just a matter of asking them what they like. All too often they will tend towards preserving their existing work environment. Somehow in the requirements process end users must be led beyond what is familiar to new approaches that will improve their capabilities.

A rather vague concept that has received much attention in recent years is the reuse of mature (that is, relatively reliable) software. The core idea is obviously correct: to the extent you reuse reliable software, your software will be reliable.26 “The most radical possible solution for constructing software is not to construct it at all.”27 This is exactly
how we work with PCs and Macs, we adapt canned software like Excel and Word. These are highly tested because of heavy reuse, and that motivates us to be willing to adapt our needs to their not entirely ideal capabilities. In HEP, canned software tools are used for histogramming (HBOOK, PAW), for farm and parallel computing support (CPS, Canopy), and for on-line/data acquisition (PAN-DA). These are all examples of large packages that are so well publicized it is hard not to be fully aware of them at times when they might be applied. Identifying smaller modules that might be reused is a more difficult problem.

The hype associated with software reuse mostly concerns the automatic cataloging of software and what it is meant to do. Automatic cataloging of software is an unrealistic promise that has been made by some in the artificial intelligence community. Clearly, if it were possible to define for a computer exactly what you wanted, and if there were a catalog of what existing software could do, all categorized well, then the problem would be solved. You would just click your mouse, and, voilà, the software you needed would be ready to reuse, in perfectly (and relevantly) documented form.

The idea behind reusing software is more subtle than the hype indicates. Two software-engineering approaches, one dating back to 1970, are important to the reusability goal. These are information hiding, a concept suggested by David Parnas in several early papers and, what can reasonably be considered its descendant, object-oriented programming. Parnas's deceptively simple idea is to have small teams of coders work on software modules. These modules communicate only through extremely well-defined interfaces. The modules contain deliberately hidden information, one or more secrets that define how the module operates. Other modules – and teams – know the interface definition but need not and should not know any but their own secrets. The secrets may, for example, refer to hardware-specific matters that are in this way isolated to single modules. In normal programs a change of hardware can have consequences throughout a huge software package. Parnas's secrets restrict consequences to single modules. Since consequences are kept local, information hiding allows easy evolution of requirements and change of software. It also is effective in allowing reuse of these very well defined and consequence-isolated modules. Information hiding encourages compartmentalized, well defined tasks, which are easier to categorize for possible reuse. It does nothing to assist in the dream of automatic cataloguing. But at least, if a module is remembered, it is likely to be reusable.

Object-oriented programming has gained notice in recent years, much of it as a result of the development of the Smalltalk language by the Xerox Palo Alto Research Center (PARC). Smalltalk was subsequently discovered by Steve Jobs on a legendary visit to Xerox PARC. It was then applied to the famous graphical human interface of Apple's Macintosh computer. Objects consist of data and one or more methods (like program procedures) that operates on the data. One communicates with objects by passing messages between them telling them what to do. The well-defined message-passing interfaces and the consequence-isolated methods of the objects are certainly within the spirit of the information hiding philosophy. A class of objects is defined once; it may have many objects that are instances of the class (such as the multiple windows on a
Macintosh). Classes, with their own specialized traits, may inherit general traits from other classes (for example, the traits of a basic window may be inherited and augmented by scrolling capabilities). One reuses what is needed and never has to describe anything twice. This is true software reuse, building on what has been done before. A large object-oriented program is built up – and tested – object by object, in an evolutionary way that clearly reduces bugs. The ease of changing objects carries with it a disadvantage from a project management perspective, the difficulty of controlling change.\textsuperscript{30}

Evolving requirements implies that software is not finished after its initial "build". This has led to the concept of \textit{incremental builds} where the ever changing nature of software is recognized from the outset. Really, as Brooks noted, programs are grown not built. This concept must be very familiar to high energy physicists whose basic approach to programming is incremental. Information hiding is exactly what one needs to support this (good) habit in an efficient way because it insists on a structure that is receptive to unanticipated changes and additions.\textsuperscript{31}

Information hiding and object oriented programming offer the possibility of a painless discipline for HEP software that fits well the incremental build approach commonplace in the field. The experience from other areas is that these techniques add only small execution inefficiencies while providing significant productivity and reliability enhancements. Professionally developed trigger, reconstruction, or analysis skeletons, within which physicist written programs could be incorporated, are natural opportunities for applying information hiding or objects. The skeleton could reach a high level of trustworthiness through some of the traditional and modern disciplined approaches we have described. The physicist code would be caged, so to speak, in regions of asymptotic freedom where the only required discipline would be at the defined interfaces on the boundary.

Another, very different, approach to increasing trustworthiness is the use of formal methods to prove (jargon is \textit{verify}) that the program does what the specification says.\textsuperscript{32} The process of being sure that the specification says what the humans want is known as \textit{validation}. The specification is written in a formal top-level specification language (FTLS) which is more "English-like" than conventional languages. After conversion, the result is code about ten times longer than the FTLS spec. Formal methods are mathematical proofs that the FTLS and the computer program are equivalent. Program source code and specification languages present the same scale of difficulties in assuring their correctness. Formal methods, therefore, are essentially amplifiers by an order of magnitude of the size of assuredly correct software segments, from about 1,000 lines to about 10,000. Though this may be extended somewhat by a hierarchical application of formally proven modules, one soon gets into the problems of measurable software reliability discussed in an earlier section. A further problem is that as the size of formal mathematical proofs increases they reach a point where confidence in the correctness of the proof itself is at issue.
Formal methods were applied by the National Security Agency's National Computer Security Center (NCSC) at the highest levels of its standards for "trusted computer systems". Here a Trusted Computing Base is verified with formal methods. Honeywell's Secure Communications Processor (SCOMP) is an experimental system at the A1 summit of the NCSC classification. Only twenty copies of SCOMP have been installed because of its limited scope and poor performance. Despite a limited scope, the formal specification for SCOMP requires the equivalent of over twenty dense pages of mathematical spec notation. This amount must be somewhere near (some evidence suggests it is beyond) the maximum that humans can produce with reasonable assurance that it is error-free. Note that even at this A1 summit of security, there is no requirement to prove that the code implemented is equivalent to the verified design! Despite these difficulties, formal methods might have applicability in high energy physics trigger software, where one can imagine developing verified trusted trigger kernels. These would be treated like the microcoded kernels in use on Level 2 triggers and called by a trigger decision language.

When considering which other modern software technologies might be able to help in improving software, one cannot neglect artificial intelligence (AI). Because of an excessive level of historical hype and unrealizable promises made in the mid 1980s, AI has been somewhat discredited. This may be an overreaction. With moderate expectations, one important subfield of AI, expert systems, does have potential relevance to high energy physics software. Expert systems are rule-based programs for tasks requiring expertise. They tend not to use deductive logic, but rather abductive reasoning. The word abduction describes the unnatural process of generating explanations, cause from effect. Normal logic proceeds by deducing effect from cause. Abduction involves such heuristic techniques as plausible inference and the weighing of evidence. Most important, for lack of a broad understanding of the world shaped by common sense, expert systems must be limited to narrow and isolated domains.33

Famous examples of the effective use of expert systems are diagnosis of breakdowns of railroad locomotives, and, in the case of Digital Equipment Corporation's VAX series of computers, determination of appropriate customer configurations. Expert systems appear to present a significant debugging problem when they get large. This implies that an inherent size limit may come into play in the future.34 The process of obtaining knowledge from a human expert (on locomotives, for example) and putting it into a form that an expert system can refer to goes by the loaded name, knowledge engineering.

Not surprisingly getting experts to agree, when there is more than one, is a significant problem. It is an interesting, perhaps more than semantic, question whether expert systems dispense knowledge or doctrine. One can imagine that when the use of medical diagnostic systems becomes widespread, any medical doctor who ignores the questions and conclusions of the systems will risk malpractice suits. Expert systems are really storehouses of doctrine.

Their creation forces a systematic analysis of information and procedures. They can provide a framework to organize and control change in large, complex research
projects. This could be applied effectively to HEP data acquisition, control systems, and software in contexts like operation, diagnosis, repair, and updates. Diagnosis in the absence of an expert (owl shift guidance on Fastbus, for example) is the classic application on experiments. One can imagine extending this to software maintenance situations such as porting code to new platforms and system releases. An expert system could be used for acceptance tests self-administered by physicists before their new code is certified for inclusion in production packages. Although expert systems are reasonably effective at disbursing expert information and doctrine, loading them with the information is not straightforward. An effective means for do-it-yourself knowledge engineering would make it possible to broaden the application of expert systems in high energy physics - - and many other fields.

I have intended, in this discussion, to give a flavor of what is unique about “the software problem” and why intuition and experience gained from other technology areas can be very misleading. The emphasis here has been on describing how the problem is being attacked. This is one good way of viewing this rather intangible subject and sensing how difficult the struggle to manage software development really is. Looked at from another direction, when computer systems are successfully applied, as they so often are, this serves to whet the appetite for applying them to more complex problems. A prime cause of the software problem is that our ability to imagine ever more ambitious systems is not limited. What is limited is our ability to understand and express, and then to produce and test, what we imagine, precisely enough to make our dreams work properly in reality.

Most important is to understand the extreme difficulty of making big advances in the productivity of writing software or the reliability of the result. There are no simple solutions, “no silver bullet” as Brooks titled a recent paper. One needs to exercise considerable judgment in walking the knife edge between anarchy and the potential for sloppy unreliability, on one side, and excessively bureaucratic philosophy constraining creativity and changeability on the other. The trustworthiness of HEP software will relate directly to the sensible use of techniques and methodologies of modern software engineering.

XII. Conclusion: Elements of an Advanced Software Engineering R&D Agenda

Several possible R&D directions in software engineering have suggested themselves as the discussion in these lectures developed They included

- Complexity estimation methods for large research projects that take into account the impact of hardware design and the sociology of research groups.

- Software project management methodologies for big science research projects in undisciplined multi-institution environments. Workable review and independent testing methodologies appropriate in these conditions.
• Integrated CASE tools for big research projects that incorporate project management and review and testing requirements.

• Painless structuring techniques for large scientific projects that apply information hiding and/or object oriented software techniques.

• Tools for easy entry of expert knowledge and doctrine into an expert system. Do-it-yourself knowledge engineering.

• Application of formal methods to create trusted trigger kernels.

Many of these problems require a study of group dynamics and software management in a large project research environment. Most, if not all, such work was previously in the context of business, computer industry, and (to a lesser extent) military software projects. As has been the case in other technology areas, it is likely that the true software engineering needs of HEP, once we look at them closely, will push the technology envelope in advance of the needs of the rest of society. These problems appear to attract considerable interest from research software engineers and organizational behavior social scientists. The result of such work will not be a silver bullet for HEP’s -- or anyone else’s -- software problem. And physicists do not recognize a software crisis at this point. Nonetheless, even small improvements in scientific productivity and the reliability of large project software are likely to be critical to the continued success and acceptance of large budget, multi-year scientific efforts. Serious consideration must be given to high energy physics driven efforts in advanced software engineering research.

Acknowledgements

I would like to thank Jim Patrick, Liz Buckley, Marvin Johnson, Jim Linnemann, Maris Abolins, Gene Fisk, Stu Loken, Irwin Gaines, Luann O’Boyle, and Nelly Stanfield for recent discussions and special help with these lectures and my colleagues in the Fermilab Computing Division for stimulation over the years. This paper draws in part on work carried out while I was on sabbatical at the Center for International Security and Arms Control at Stanford University and uses some material published in a working paper by the Center.37 I was supported at Stanford in part by the Carnegie Foundation through a Carnegie Science Fellowship. Fermilab is funded by the Department of Energy.
References


17 At this mention of "software engineering", it is important to remind that "engineering" here does not carry with it the usual implication that we are referring to a quantitative discipline, like, say, "mechanical engineering". We do not know how much stress will cause a software structure to collapse. In fact, as I try to show here, we do not even know what such a question means in any quantitative sense.


19 One should note, as David Weiss comments, the significant absence of a production phase in software technology.

22 Private communication 1989.
30 For more on object oriented programming, see Paul Kunz's lectures at this summer school.
33 More details on AI and expert systems may be found in the lectures by K.H. Becks at this summer school.
34 David Weiss, private communication, August, 1989.
Figure Caption

Figure 1. Conceptual data flow and triggers for the Solenoid Detector Collaboration (SDC) system at the Superconducting Super Collider (SSC).
SIMD Processor Arrays

D. Parkinson

Computing Centre, Queen Mary College, London

INTRODUCTION

Parallel Processing can be defined as the use of many processors simultaneously to solve a single task. That definition leaves open the question as to what is a task in the scientific community a task is usually some self contained job such as "simulate some complex process", Solve a set of equations etc. The major characteristic of a task for our purposes is that there is no easily identified subtask which can be solved in isolation.

Parallel processing uses a number of processors to solve the task and these processors must cooperate by sharing data and partial results to complete the total task.

The world of parallel computing is apparently split into two camps. The SIMD community and the MIMD community.

The SIMD (Single Instruction stream Multiple Datastream) community studies computer organization which consist of Master processor and an number of subsidiary processors (processing elements or PEs). The master processor is the sole source of instructions which it send simultaneously to each of the subsidiary processors. An instruction might be considered to be of the form:

Add the contents of location 5 in some local memory to location 12 in some local memory and store the result in the 17th location.

If there are say ten thousand subsidiary processors the effective computation rate will be ten thousand times faster than the single processor. The crucial architectural feature of the SIMD world is this tightly structured set of strictly synchronized processors. The MIMD world is less strictly controlled.

The MIMD (Multiple Instruction Multiple Datastream) community studies computer organizations in which there is usually no master processor and so each individual processing element has a private program which it runs at its own speed. The system is therefore not synchronous like the SIMD systems. MIMD systems are "obviously" more flexible than SIMD systems at the cost of the need to provide a synchronization mechanism which ensures that two processors which need to communicate are both ready and willing to do so.

It is not my intention to give here a detailed evaluation of the pro- and cons of SIMD systems versus MIMD system that is for two reasons:

a) I am strongly prejudiced in favour of SIMD systems,

b) The organizers have asked me to concentrate on SIMD.
Some of the relative merits of the two approaches are summarized below.

**SIMD Systems**

**Advantages**

a) Currently manufactures can build very much larger and more powerful systems using the SIMD approach.

b) Synchronization of processors is automatically ensured by the hardware.

c) Only one copy of a program need be stored.

e) Parallelism across DATA structures is easily exploited and provides a ready source for exploiting ever larger arrays.

f) Higher efficiency of utilization of Large arrays.

**Disadvantages**

a) Requires a unfamiliar programming model and data parallel operations cannot be easily extracted for serial code.

b) Some processors are inevitably idle waiting for others to finish, thus reducing apparent processor utilization.

c) Entry level systems have at least 1000 processors and so the use of and SIMD system requires a large initial commitment.

**MIMD Systems**

**Advantages**

a) Easier/cheaper entry to systems as one can, in principle, start with two processors and slowly build up

b) Greater flexibility of coding strategies.

c) Easier translation of serial code onto a few processor environment.

**Disadvantages**

a) Large potential overheads when systems have to synchronize prior to communication

b) Increased programming difficulty as the number of processors gets greater.

c) Rapidly decreasing overall utilization as number of processors increases.

d) Less well developed software technology.
For the rest of this lecture I shall concentrate of the description of some commercially available SIMD systems but first I want to discuss some general aspects.

SIMD PROCESSORS SYSTEMS IN GENERAL

The prime concept of the SIMD processor system is that of a single master processor which issues a stream of instructions to an army of slave processors (the processing elements). Important hardware variables in the system are:

1) The number of processing elements,
2) The characteristics of the processing elements,
3) The interprocessor communication mechanisms,
4) The amount of memory per processor.

There is inevitable trade-off between the number of processors and the power of the processing elements. For a given build-cost, the more powerful processor that is used the fewer that one can use. The cost of processors is not linearly related to their power so it is not necessarily the case that the maximum system power is obtained by using a few powerful processors. The trade-off is further complicated by the difference between the theoretical maximum power and the actual delivered power. If we had a universally accepted definition of computer power we could compute a theoretical maximum system power by multiplying the power of a single unit by the number of processors. The delivered power of any computer system is usually less than the maximum power - that performance the salesman guarantees you will not exceed! Different computer systems produce different fractions of the maximum power and the factor also depends strongly on the application. If such a thing as an honest salesman existed, it might be possible to find an application demonstrating the maximum power but usually the performance is much less.

In any reasonable application of a parallel processing system there are addition factors which affect the total performance the most important of which is the cost of inter-processor communication. To understand some of the considerations we will look at a trivial problem.

Suppose we have been performing a computation on an 8 processor system and have generated a number in each of the processors. The application now requires that we find the total of the eight numbers.

A High level language programmer finds no difficulty in this task they just write something like

```
sum=0
di 100 i=1,8
sum=sum+x(i)
100 continue
```

The strategy implied in the above code is very serial in nature, and only implies the use of one arithmetic unit.

There is a well known parallel way of doing this task as shown in Figure 1.

Three parallel steps are needed. In step 1 partial sums of pairs are needed. In step 2 the pair sums are added in pairs to give two sums over four elements and the final step uses one processor to produce the final sum of eight elements.
In general for adding $N$ elements $\log_2 N$ steps are needed. It is therefore not unusual to see statements that the time for the summation is $\log_2 N$ multiplied by the addition time for a uni-processor. There is, additionally, some cost involved in moving data between processors. If we assume that the processors are linearly connected, as in Figure 1, we see that the first step required data to move a distance 1. The second step requires data to move a distance 2 and the third step data moves a distance four. so the accumulated data movement has the cost for 7 movements. This result is no surprise because the problem requires that somehow, data 7 locations apart are brought together. In general on a linear array there will be $(N-1)$ movements.

The total cost for our simple summation algorithm is therefore

$$\log_2 N \ T_{add} + (N-1)T_{move}$$

Which term will dominate depends very strongly on the ratio of $T_{add}:T_{move}$ and $N$.

The analysis is only valid if the processors are connected in a linear array. If we took the ideal case in which every processor was connected directly to every other processor the result would be

$$\log_2 N \ (T_{add}+T_{move})$$

From this trivial example we see that when we are evaluating a multiple processor system it is not sufficient to look purely at the arithmetic capability of the processors (ie $T_{add}$) but to example the costs of the interprocessor communication ie ($T_{move}$ and the communication network).

**INTERPROCESSOR NETWORKS**

The ideal interprocessor network is one in which every processor is directly connected to every other processor and for which there is no cost to initiate a transfer and for which the transmit speed is infinite. Such a network is somewhat impractical and it is important to discuss real networks. There are hypothetical
parallel computational models which ignore interprocessor communication but such models seldom have any relevance to practical systems.

The study of communication networks is a very large field and it is found that different types of networks are optimal for different types of application eg, a tree structured network is very well suited to operations such as the summation operation considered above. The task of the designer of system aimed at a wide range of applications is to choose a network which represents a trade off between cost (the more connections the more it costs), performance (the longer the wires the slower it runs) and generality.

![Diagram of mesh connection schemes](image)

**LINE**

**NEWS**

**8-NEIGHBOUR**

Figure 2: Mesh Connection Schemes

In practice any network which provides a route from any processor to any other processor can simulate any other network, the difference is only a matter of speed. A simple 1 or 2 dimensional network can be laid out on a chip or board with very short wire lengths. higher dimensional networks inevitably have a long wire and so must clock relatively slower.

There are very few networks which have been are used in commercial systems Figure 2 shows some popular mesh topologies. The two dimensional scheme (NEWS) is popular as the cost is not much greater than the linear scheme and for N processors reduces the longest distance between two processors from N-1 to 2($n^{1.5-1}$). In practice the system usually are implemented as circles and toroids for little additional complexity.

The NEWS network is well adapted to working on problems defined on two dimensional square meshes eg: partial differential equations, image processing matrix algebra etc.

There are problems which have different connectivity than four neighbours. Some image processing algorithms deal with an eight neighbourhood scheme, whilst some applications have a triangular or hexagonal mesh. The eight
neighbour mesh is popular with systems designed for such schemes as it effectively halves the transmission times between say the NW neighbour and the central processor.

In recent years the Hypercube interconnection scheme has become very popular.

Figure 3 shows the first 3 hypercubes. The task of drawing hypercubes connection diagrams is one of the ways the current generation of computer architects enjoy themselves. The pictures are not quite as pretty as Fractal sets.

![Hypercube schemes](image)

**Figure 3: bHypercube schemes**

**SOME CURRENT SIMD SYSTEMS**

There are three widely available commercial SIMD systems

In historical order of these are:

DAP now sold by Active Memory Technology AMT,

The Connection Machine (CM) sold by Thinking Machine Inc (TMI) and

MASPAR machine from Maspar Corporation.

This is not the place to give a detailed technical evaluation of each other these systems I shall restrict the description to the salient Hardware and Software characteristics. There are a number of common properties, the first of which is a mechanism by means of which only a subset of the processors obey an instruction. This feature, which I shall call Activity Control, is absolutely necessary to provide sufficient application flexibility. In principal boolean operations could be performed to simulate such a facility but it has proved more efficient to provide direct hardware facilities.

Many users express a desire to use old "dusty deck" Fortran programs on these machines. Unlike the vector processors world, the Parallel computing world has found no way to automatically translate( efficiently) old serial codes and make them run on their machines. None of the manufacturers claim efficient tools for automatic parallelisation of old programs.
All three systems have an associated Front end machine which is usually responsible for the operating system, file store compilation etc.

All the systems have processing elements with short word lengths, 1 and 8 for DAP, 4 for the Maspar and 1 for CM2. Some people express unnecessary worries about these lengths, however all systems are programmed in high level languages which completely isolate the user from word length constraints. Fortran always works on Fortran type variables. The low precision nature of the processing elements gives a benefit that low precision or logical operations work relatively faster than on long word length hardware.

AMT-DAP systems

The DAP systems are have either 1024 or 4096 processors. The first generation of DAP systems had very simple 1 bit processors the recent systems have 1024 or 4096 processor pairs, each pair consisting of a single bit processor and an 8 bit co-processor.

The associated Front end system which can either be a SUN Unix host or a VAX VMS host.

The one bit Processing elements are 64 on a chip with a 1024 processor system only requiring 4 boards for the total processor array and memory. DAP machines are physically small a 1024 processor machine fitting under a desk and a 4096 processor machine being the size of a filing cabinet.

The Interprocessor network on the DAP systems is the NEWS network discussed earlier. Additionally there is a "Row and Column Highways system" which is a set of data paths one per row and column which can be used to globally broadcast data from a central register held in the Master processor. These Highways can also be used to test the array for operations such as Maximum element. The programming system is an Array extension to Fortran similar to FORTRAN 90. A wide range of library routines are available.

The performance range of the systems is up to a maximum of 500 MFlops for a 10 Mhz 4096 processor DAP.

CM-2

The CM-2 systems have 4k, 8K, 16K, 32K or 64K processors and these 64K processors systems are currently the largest commercially available parallel processing systems. Some experimental image processing systems have been built with 256K processors.

The processing elements are 1 bit processors although associated with every 32 processors is a Weitek floating point chip. The difference between the CM-1 machines and the CM-2 machines was the addition of this floating point chip. The 64K PE CM-2 may therefore be considered as a either a 64K machine of 1 bit PEs or a 2K machine of 32 bit PES. This ability to view the same hardware in two different ways is common to all the systems we are discussing and is a trap for those people who want to indulge in easy characterization. The theoretical performance of 2K Weitek chips is in excess of 3Gflops and some applications are approaching this performance.

Typical front end systems for CM2 are VAX or Symbolics machines.

The interprocessor network is a hypercube with up to 12 dimensions for the largest systems. The processors themselves are 16 per chip and form a local news network. An very important feature of the interprocessor system is the Router
network which allows the system to be used for arbitrary data routing. A processor can give the Router network the address to which it wishes to transmit some information and then let the network find an appropriate route. Naturally such operations are much slower than regular routing over the hypercube.

Programming for the CM-2 was originally in a dialect of LISP reflecting the ancestry of the Machine in the AI community. More commonly the machine is now programmed in C* a C dialect with data parallel extensions, or FORTRAN 90.

Physically the CM systems are built around stand alone cubes with a rather famous set of pretty flashy lights.

Maspar

The Maspar systems

Maspar System systems have either 1K, 2K, 4K, 8K or 16K processors

The processors are 4 bit nibble processors with extra support for floating point operations. The theoretical maximum performance of the 16K machine is 1.5 Gigaflops for 32 bit arithmetic. Host are restricted to DEC systems running Unix. There is little information yet available on actual performance but that information which does exist suggests that the delivered perform very much less than the theoretical.

A very attractive feature of the Maspar processing element is the existence of an independent index register in each processor. The index register, missing in DAP and CM-2 allows greater flexibility in local addressing schemes.

The interprocessor network for the Maspar is the eight neighborhood mesh (called X-net ) by Maspar. There is also a equivalent of the Routing network function discussed in the CM2. The bandwidth of router networks is usually an order of magnitude less than the bandwidth of the simpler networks and therefore they are only useful for highly irregular interprocessor movement.

Programming is either in a variety of C or a Fortran 90 variant.

Maspar system the size one or two filing cabinets.
VIRTUAL MACHINES VERSUS REAL MACHINES

In the previous lecture we discussed SIMD processors. The basic model of an SIMD processor is that of a master processor which issues the same instruction to each and every one of an array of identical processors. In the user view of such systems it is desirable that the operating system/language isolates the users from an exact knowledge of the number of real processors and so it is usual for people to now work with a model in which they have access to a virtual machine which has exactly the correct number of processors for their problem. The term Virtual Processors was first used by TMI when they introduced a programming model that allowed the user, at compilation time, to specify the virtual size of the machine that they wanted to use. A user programming a problem on 256*256 arrays would ask for a virtual 256*256 machine and then program as if all arrays where mapped element by element into such a machine. Calling three such arrays A, B and C the simple FORTRAN statement

\[ A = B + C \]

would cause corresponding elements of B and C to be added and the result placed in the corresponding location of A. The compiler/runtime system would map the 256*256 elements of each array into the real machine. Suppose the real machine has 4096 processors, the compiler allocates 16 elements of each array into any given real processor. Alternatively one can view the operation as request for each actual processor to simulate 16 virtual processors.

The original TMI virtual machine was a static concept but modern systems, such as embodied in the FORTRAN 90 language allows the Virtual machine to dynamically change as the data structures being manipulated change in size and shape.

A programming model which releases the user from knowledge of the actual physical attributes of the machine has the advantage that the user can program with a greater degree of freedom. The disadvantage is that large fluctuations in performance will be experienced for apparently small changes in problem size.

We can see how these fluctuations come about by examining how a 64*64 processor array will execute the code to add two matrices.

Using Fortran definitions we can define

real a(N,N),b(N,N),c(N,N)

and examine the execution of the statement
A = B + C

A totally reasonable strategy is the map the arrays one per processor. If N is less than or equal to 64 the arrays only occupy one location per processor and the all the addition take place simultaneously giving an execution time of $T_{add}$. If however N is between 65 and 128 it is probable that the arrays will be broken into four parts and up to four numbers allocated per processor.

The execution time for all N in the range 64 to 128 will then be 4 $T_{add}$. The user of such a program would see a large change in performance N changed from 64 to 65. Notice that the performance is then constant until N changes to 129. This very non-linear performance is characteristic of all parallel computing systems and is a natural consequence of the multiple processor approach. The non-linear performance characteristics means that users of parallel systems who are interested in maximum performance learn to choose their problem sizes to be exact multiples of the hardware sizes. If we stay with our trivial matrix summation task we see that the cost of adding 128*128 matrices is the same as adding 65*65 matrices, thus a larger problem can be done in the same time as a 65*65 problem. In many scientific/simulation problems problems the data structures are user definable with more accurate results being produced the larger the data structure. As parallel processors have $2^n$ processors one find that it will be more common to do 128*128 problems rather than 100*100 problems.

The programming model which allows us to assume that we have just the correct number of processors has many advantages for the user. The view can be taken that the parallelism is that of the data structure ($N*N$ in our example) and for this reason the term DATA PARALLEL computation has been coined to cover the programming model.

The data parallel model is ideal for SIMD processing and produces its parallelism by consideration of the parallelism in manipulation of the data structures being used in the application. As applications get larger the data structures tend to get every larger so there is a rich mine for the users of parallel systems. If one wishes to perform computations of 4 dimensional meshes with each side 64 one has a potential degree of parallelism of $64^4$. In theory one would like $64^4 = 16$ million processors for such a problem. When applications are studied from the data parallel point of view it is commonplace to discover that the 'Ideal' number of processors is much greater than the actual number and there is scope for optimization in mapping the large degree of parallelism in the real problem onto the limited degree of parallelism of practical systems. In next lecture I discuss one such a problem.

First we want to concentrate on more general parallel algorithms and core operations.

**SIMPLE PARALLEL ALGORITHMS**

The matrix addition task discussed above is representative of a large class of parallel operation in which element by element operations are performed between identical operations in two data structures. 75% of most parallel
processing on SIMD arrays is of this nature and is so simple and banal that we cannot find more to say about it.

More interesting problems arise when we want to discuss problems on meshes and neighbourhood operations on meshes. Such problems arise in many disciplines most typically in the solution of multi dimensional partial differential equations or image processing. From the Data parallel viewpoint there is not much difference between many of the operations of image processing and partial differential equations. Most operations in these disciplines may be conduced as the evaluation of weighted sums of a set of elements covering some neighbourhood of each point.

A typical operation might be to compute an approximation to a second derivative by the formula:

\[ D_{2i,j} = x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1} - 4.0 \times x_{i,j} \]

If we assume that element (i,j) is in a virtual processor (i,j) we see that the task requires that data elements are moved between processors. The data parallel approach interprets these operations as 'Shifts' applied to the data arrays in exactly the same way as one applies shifts to words in registers of a single processor.

If we think of a two dimensional array as laid out on a sheet of paper we can consider shifts to be UP, DOWN, LEFT or RIGHT we can create some pseudo-code for the above formula

\[ D2 = \text{SHIFT\_UP(X)} + \text{SHIFT\_DOWN(X)} + \text{SHIFT\_LEFT(X)} + \text{SHIFT\_RIGHT(X)} - 4.0 \times X \]

Where we are assuming that SHIFT\_UP(X) is an operator which returns a data array which is equivalent to that which would be obtained by shifting all elements X up one row. The alert reader will have two questions

a) Where is the origin of coordinates?
b) What is happening at the edges?

The answer to the first question is – For this problem is does not matter!

Because the original problem was symmetric, the code is correct no matter what the operators assume for the origin of coordinates. Of course, any language implementation must have a well defined interpretation but the symmetric case is common. It is not unusual that parallel coding is independent of such fine detail.

The question about the edges is more interesting as a full definition of the operators must specify what happens at the edges. The FORTRAN 90 dialect has two operators of the SHIFT type. These operators are called EOSHIFT and CSHIFT (being FORTRAN they are called FUNCTIONS).

EOSHIFT performs and End-Off shift of an array. Values are lost at one end of the array and new values are introduced at the other end. The default is to introduce zero but a boundary value may be specified.

CSHIFT performs a cyclic shift in that values which are shifted off at one end are introduced at the other.
Hence if our space was the same shape as a soft drink can i.e. cyclic east-west and planar up down the FORTRAN 90 code would be

\[
D2 = \text{EOSHIFT}(x,1,\text{Boundary}=0.,\text{dim}=1) + \text{EOSHIFT}(x,-1,\text{Boundary}=0., \text{dim}=1) + \text{CSHIFT}(x,1,\text{dim}=2) + \text{CSHIFT}(x,-1,\text{dim}=2) - 4.0 * X
\]

(I do accept responsibility for the ugly conventions of FORTRAN 90)

You should note that this single line of code replaces many lines of normal FORTRAN and is independent of the declared dimensions of the arrays. Provide X and D2 have been declared to be conformal the code is correct for arbitrary array sizes.

Another important thing to note about this style of programming is that there are no indices in the code. Many years of teaching effort have been expended in teaching generations of programmers to use indices, or pointers, as means of addressing elements of data structures. We now have a retraining programme where we have to teach people to think of objects in their entirety rather than the result of iterating sets of indices.

Another observation about the above FORTRAN 90 fragment is the fact that the boundary conditions are included within the 'main loop' of the code and are not included in some special purpose coding. In our example the boundary conditions are elementary and could be handled by the basic operators of the language. In more complex examples other techniques will be needed. In serial programming it is common practice to identify special cases and write code which identifies the special cases and executes special code. Parallel processing is more 'pure mathematical' in spirit and good practice is to try and try and cast all cases into a general case with the data handling the special cases.

To show the principles suppose we are solving an elliptic equation on the surface of our soft drink can.

For clarity I will use a pseudo-code rather than Fortran.

Suppose we have five arrays UP, DOWN, LEFT, RIGHT and HOME the line of code

\[
X = \text{UP} * \text{SHIFT_UP_CYCLIC}(X) + \text{DOWN} * \text{SHIFT_DOWN_CYCLIC}(X) + \text{LEFT} * \text{SHIFT_LEFT}(X) + \text{RIGHT} * \text{SHIFT_RIGHT}(X) + \text{HOME} * X
\]

would code a general purpose iteration scheme which can be made to cover many special cases. For example if there are points where X should be held constant then at those points UP = DOWN = LEFT = RIGHT = 0 and HOME = 1, similarly constant gradients etc can be imposed through proper choice of the parameters. In this programming style it is the DATA which is being used to control the operations rather than a special program. For some people there is a major culture shock involved when they have to come to terms with the fact that multiplying by zero is a good strategy! In a parallel world it is not the total number of multiplies that is important it is the number of 'sets' of multiplications that are important.

Recursion Problems
Although 90% of scientific computation can be covered by the techniques we have discussed there is a class of computational problems which, at first glance appear unsuitable for parallel computation. Probably the simplest example of such tasks arise from tasks specified by recurrence relations.

For example

\[ x_i = a_i x_{i-1} + b_i \quad \text{for } i = 1, \ldots, n \]

where the set of number \( a_i \) and \( b_i \) are given and some starting value for \( x_1 \) is specified.

Beginners think that such problems cannot be solved in parallel "because you must compute \( x_{10} \) before you can compute \( x_{11} \).

We however have seen in the previous lecture a simple version task. If we have the case \( s_i = 1 \) for all \( i \), the solution for \( x_N \) is the sum of all values of \( b \) up to the \( n \)-th. The addition tree solves the problem in \( \log_2 n \) steps.

The majority of the special parallel algorithms are of this nature in which a problem which at first glance looks as if it need \( N \) steps is solved in \( \log_2 N \) steps.

The technique is powerful and should be understood by all programmers of parallel systems.

There are many ways of presenting the method, I shall use a algebraic approach.

Our original recurrence relation is, by definition valid for all values of \( i \)

hence \[ x_{i-1} = a_{i-1} x_{i-2} + b_{i-1} \]

If we substitute this relation into the original relation we can derive the equation

\[ x_i = (a_i a_{i-1}) x_{i-2} + (b_i + a_i b_{i-1}) \]

As stated before the recurrence relation is defined by the coefficients sets \( a_i \) and \( b_i \). Which If we replace the \( a_i \) by \( (a_i a_{i-1}) \) and \( b_i \) by \( (b_i + a_i b_{i-1}) \) we have converted the original problem into a new problem which now connects \( x_i \) with \( x_{i-2} \). This conversion is an operation on the defining data sets. The algebraic process may be repeated again and a new coefficients computed which specify a relationship between \( x_i \) and \( x_{i-4} \). \( \log_2 N \) steps are needed to cover the range \( i = 1, \ldots, N \).

The recurrence relation can therefore be solved in \( \log_2 N \) steps each step having two multiplies and addition and a shift. \( \log_2 N \) is a slowly varying function of \( N \) and for all intents and purpose is a constant (of order 10) for practical values of \( N \).

This technique is a powerful tool whose understanding is critical to the efficient solution of some problems in parallel computing.

Tridiagonal Linear equations.

One instructive example of the techniques discussed in the previous section occurs when solving tri-diagonal linear equations. An efficient tri-diagonal solver is a useful tool for many applications. Parallel algorithms for solving tri-diagonal linear equations have been extensively studied, Hockney and Jesshope[1] is a good
source for discussion of the problem. The optimal parallel algorithm for solving
one set of tri-diagonal linear equations is the Cyclic Reduction Algorithm and
uses N processors.

Our concern here is as much with programming as with theoretical analysis,
so we consider a tri-diagonal set of linear equations as being defined by four one-
dimensional arrays A(N), B(N), C(N), D(N) which define a set of tri-diagonal
equations which we write algebraically as

\[ b_i x_i = a_i x_{i-1} + c_i x_{i+1} + d_i \quad i=1 \ldots \ldots n \]

with \( a_1 = c_n = 0.0 \)

The cyclic reduction algorithm transforms these equations into a new set by
the operations

\[ A_i = a_i / b_i \]
\[ C_i = c_i / b_i \]
\[ D_i = d_i / b_i \]

then

\[ a_i' = A_i A_{i-1} \]
\[ c_i' = C_i C_{i+1} \]
\[ b_i' = 1 - A_i C_{i-1} - C_i A_{i+1} \]
\[ d_i' = D_i - A_i D_{i-1} - C_i D_{i+1} \]

The equations we now have are

\[ b_i' x_i = a_i' x_{i-2} + c_i' x_{i+2} + d_i' \]

The algorithm repeats this transformation \( \log_2 n \) times to produce the result.

Programming the algorithm in FORTRAN 90 is instructive since the
operations all take place on the data structures a, b, c, d. We can write the inner
loop as

```
  K = 1
  DO 1 ISTEP = 1, LOG2N
    A = A / B
    C = C / B
    D = D / B
    B = 1 - A * EOSHIFT(C, -K, DIM = 1) - C * EOSHIFT(A, K, DIM = 1)
    D = D - A * EOSHIFT(D, -K, DIM = 1) - C * EOSHIFT(D, K, DIM = 1)
    A = A * EOSHIFT(A, -K, DIM = 1)
    C = C * EOSHIFT(C, K, DIM = 1)
  1 K = 2 * K
END DO
```

The solution is \( D / B \).

An alternative is to use the WHILE construct.
K=1
DO WHILE K.LT.N
  A=A/B
  C=C/B
  D=D/B
  B=1 -A*EOShift(C,-K,DIM=1) -C*EOShift(A,K,DIM=1)
  D=D -A*EOShift(D,-K,DIM=1) -C*EOShift(D,K,DIM=1)
  A=A*EOShift(A,-K,DIM=1)
  C=C*EOShift(C,K,DIM=1)
  K=2*K
END DO

The benefit of this version is the elimination of the need to compute the variable LOG2N.

One way of describing the functioning of the cyclic reduction algorithm is to consider that each step doubles the number of zeros between the central diagonal and the off-diagonal terms. After log₂N steps the off-diagonal terms disappear (become all zero) and we can alter the program to reflect this view to the following:

K=1
DO WHILE ANY(A.NE.0.OR.C.NE.0)
  A=A/B
  C=C/B
  D=D/B
  B=1 -A*EOShift(C,-K,DIM=1) -C*EOShift(A,K,DIM=1)
  D=D -A*EOShift(D,-K,DIM=1) -C*EOShift(D,K,DIM=1)
  A=A*EOShift(A,-K,DIM=1)
  C=C*EOShift(C,K,DIM=1)
  K=2*K
END DO

As Hockney and Jesshope point out one can, if the equations are diagonally dominant, consider replacing the tests versus zero with a test versus some tolerance factor. The code we have produced has a more surprising property when we realize that there is no explicit reference to the number of equations being solved! At first glance this may seem to be a minor quirk with no great significance, but it hides an observation of much greater significance.

The code that we now have will not only solve one tri-diagonal system it will solve any number of tri-diagonal systems of any combination of lengths, and
the number of times that the loop will be traversed is given by the logarithm of the size of largest system in the set!

Although we originally set out to write an algorithm to solve a single instance of our problem we have developed an algorithm which solves multiple instances of the same problem. An algorithm with this property is a SUPER PARALLEL algorithm.

A given tri-diagonal system is defined by the four data structures a,b,c and d. A set of tri-diagonals is usually considered to be defined by sets of these data structures. We can however concatenate all of these small sets into a grand set; typically, in diagrammatic notation, we would have a matrix of the structure shown in Figure 1.

This structure has an overall tri-diagonal structure but does in fact define three distinctly separate tri-diagonal systems with 5,6 and 4 elements respectively. Applying the general algorithm to the above grand matrix solves the 3 subsets after the loop has been traversed three times.

```
  XX
  XXX
  XXX
  XXX
  XX
  XXX
  XXX
  XX
  XXX
  XXX
  XXX
  XX
  XXX
  XX
```

Figure 1

At first glance the result may seem like magic, but a little thought shows that we should have expected the effect. The first step of the cyclic reduction algorithm splits the input problem of solving a system of N equations into two separate systems of about N/2 equations, one set of equations connecting the even numbered unknowns and the other system connecting the odd numbered unknowns. The parallel cyclic reduction algorithm does not exploit this reduction but solves the two distinct sets simultaneously. It is therefore no great surprise that the algorithm handles multiple sets of equations.

One advantage of the above approach comes from the observation that tri-diagonal systems seldom occur in single sets but occur naturally in multiples.
Solving multi-dimensional partial differential equations by ADI methods naturally gives rise to sets of tri-diagonal systems.

It is not difficult to find other examples of Super Parallel examples; most recursive doubling type of operations are ideal candidates. Hence we can use the above methods to solve sets of recurrence relations of the type

\[ x_i = a_i x_{i-1} + b_i, \quad x_i = (a_i x_{i-1} + b_i)/(c_i x_{i-1} + d_i). \]

The characteristic of these types of operations that we are exploiting is the self-contained nature of the definition, or the possibility of defining the boundary conditions as part of the problem.

REFERENCES
Factorizing Large Integers
An exercise in Parallel Computation

D. Parkinson
Computing Centre, Queen Mary College, London

THE HISTORY
For this lecture I have decided to discuss a program conversion task. Although the ideal way to start a parallel programming exercise is from a statement of the problem it is much more common to start form an existing program. The exercise I want to discuss started when received a Fortran Program written to factorize 63 bit integers. This program implemented a well known strategy for factorization. The method used is described in (Knuth SemiNumerical Algorithms Pages 371-374).

The exercise is of particular interest as it shows how a parallel processing strategy can be almost the inverse of a serial processing strategy. I have also chosen this exercise because there is almost no specialized knowledge needed to understand the operations involved.

The Serial Algorithm
The algorithm implemented in the basic program had four main Phases.

Phase 0 Initialisation
Compute the table of all prime numbers less than 2^21. This table has 155611 entries let us call these values p_i.
For each number x to factorize, perform the following steps:

Phase 1) Primality Check
Check if x is prime if so halt.

Phase 2) Eliminate small factors
For i =1 to 155611 check x to see if p_i is a factor.
If p_i is a factor, replace x by x/p_i if x is not unity go back to step 1.

Phase 3) Factorize Large Composite number using Fermat Method
If x is greater than unity and no p_i is a factor then we know that x must be greater than 2^42 and since it has no factors less than 2^21 it must be the product of two large primes each of which is greater than 2^21. A sieve method is used to identify factors. We will describe the particular sieve method later.

Let us look at these steps in their serial and parallel implementations.
Computing all Prime numbers smaller than some $N$.

This is a well known task usually set as training example for young computer science students. The technique used in the serial program was THE SIEVE OF ERATOSTHENES. The method (page 394 of Knuth) starts with a table containing $N$ locations. Each location contains a boolean showing if the corresponding value is prime or not. Every second location is 'crossed' out. Then cross out every third location etc. At any stage we know that all values that have not been crossed out but are less than the square of the last prime used are themselves prime. Stop when we have dealt with the first prime greater than $n^{**0.5}$. In our case we would have to apply this "sieving" procedure for some 230 primes.

To quote Knuth this sieving algorithm is 'directly suited for machine computation using no multiplication'. The algorithm can be implemented easily in a serial environment using an address pointer that is incremented to give the next location to be "crossed out".

The 'crossing out' is viewed as an operation in address computation is serial in nature and so does not translate directly into a method.

We can however derive a parallel algorithm very easily. Set up a table of all the numbers from odd numbers from 1 to $N$. If we had enough processors we could assign these numbers one per processor. In every processor, test if the content is divisible by 3. If so mark that processor as containing a non-prime number. Now check for divisibility by five etc.

The parallel algorithm may, if the reader wishes, be considered to be identical with that of ERATOSHENES or may be considered to be entirely different depending on what one considers to be the crucial feature of the old method.

A few features of the parallel method are important.

1) The method does not care about the order of the candidates amongst the processors. Providing one has a method of selected the smallest unmarked number and for broadcasting a scalar to all processors one can easily implement the algorithm. One does need to be able to initialize the processors.

2) As outlined the algorithm implies that one has $N/2$ processors.

It is improbable that one will ever have the correct number of processors. One will always have fewer processors and though one may be using a software system that allows the programmer to simulate the having more processors a speed up can be obtained by if one uses packing techniques to reduce the number of "virtual processors". In the DAP implementation we first implemented the crude algorithm using a data set of candidates with elements chosen to that the only contained numbers of the form $30N+i$ with $i = 1,7,11,13,17,19,23,29$ (all other values of $i$ leading to numbers with factors 2,3 or 5). After each 'crossing out' we counted how many values we had rejected. When the number of candidates crossed out had rise to at least 128000 we packed the remaining elements into a
shorter set and carried on testing these reduced set. Experimentally we found that the cost of reduced computation more than compensated for the time taken in packing data.

**Primality Test**

As we have 155611 factors in our database a serial computation with up to 155611 divisions is apparently necessary to check if the number has one of these as a factor. This test could take a long time and so it is usually a good idea to run a test to verify that the number really does have factors. There are well documented tests that can be used to provide such information the test due to Solovay Strassen (SST test) was used in the original program. The SST test (Knuth page 396) test is based upon the computation of the Jacobi symbol defined by the equations

\[
\left[ \frac{p}{q} \right] \equiv \left( \frac{p^{(q-1)/2}}{modulo q} \right)
\]

when q is prime.

\[
\left[ \frac{p}{q} \right] = \left[ \frac{p}{q_1} \right] \ldots \left[ \frac{p}{q_t} \right]
\]

when q is the product q_1 \ldots q_t of t primes.

The relevance of the Jacobi symbol lies in the fact that it has value 1 if p is a quadratic residue mod p, is -1 if p is a nonquadratic residue mod q and is zero if q divides p.

The Jacobi symbol may be computed efficiently, with very little more work than that involved in computing the greatest common divisor of p and q. An algorithm for computing the Jacobi Symbol is described in Knuth’s text. The SST test computes

\[
\left[ \frac{x}{n} \right] and x^{(n-1)/2} (modulo n)
\]

if these are equal for some x then n is definitely not prime. The test therefore evaluates these functions for a number of values of x. The original program used a database of 32 values of x (actually the first 32 primes). Only numbers that fail the primality test are tested for factors.

Due to the computational efficiency of the test it is a good strategy, in a serial environment to apply the SST test before searching for factors from the database and after each time a factor is removed from the target number.

In an ideal parallel environment with say 155611 processors we can simultaneously test for divisibility by every element in the database and therefore in only one 'division time' test all numbers less than 2^{42} to see if they have factors. This single parallel division being much faster than applying the test. Thus in a parallel environment we find it more efficient to use the SST test only on numbers greater than 2^{42} for we have either completely factorized smaller numbers or shown that they have no factors less than 2^{21}.
As we discussed above, in real life we will not have the same number of processors as demanded by the 'ideal algorithm' theory but usually have far fewer. For example all our experiments where done using a 4096 processor system. Even then it turned out to be most efficient to perform the test divisions first. If we had only had say 10 processors we would have found it more efficient to the SST test before searching for factors.

There are two main lessons to be learnt.

The optimum algorithm for any given task is a function of the number of processors. A theory that attempts to derive optimal parallel algorithms from an assumption that one has an infinity of resources is doomed to failure. The actual cut off point between the two strategies is very hardware dependant being a complex function of arithmetic times and interprocessor communications times.

A parallel algorithm can often be based on the reverse strategy to that of the serial algorithm. We saw this before in our discussion of initial sieve to compute the table of primes. We also see that as a total problem proceeds the theoretical optimal number of processors changes.

Our parallel algorithm has not avoided totally the necessity of the primality test it has just moved it to another place in the program flow.

The primality test can be applied in parallel using one processor for each trial value. If we have lots of processors we can use them all to provide an even firmer test. The original program used a base of 32 values of x. To replicate the accuracy of the original program we theoretically only need 32 processors. Here we have a case where we have too many processors! We can however apply the test using more values of x and be even more confident that the number is really prime. The theory of the SST test suggests that it might falsely report that a given number is prime with probability less than 1/2. Therefore applying it 32 times given a probability of an incorrect result as approximately 1 in $10^8$ whilst applying it 4096 times gives a truly astronomically small chance of a false result.

**Phase 3 Fermat's Sieve method for finding factors**

The sieve method to factor N attempts to find a pair of numbers P and Q such that $P^2 \cdot Q^2 = N$ then $N = (P+Q)(P-Q)$ which is the factorization, provided $P-Q$ is not unity.

If we have such a pair of numbers P and Q then

$P^2 - N = Q^2$, with $P \gt N^{1/2}$.

Let $Y$ = the next integer greater than $N^{1/2}$, then there is an integer $M$ such that

$(Y+M)^2 - N$ is a perfect square. What we are going to do is to look all numbers of the above form for $M=1 \ldots$ and find those that can be perfect squares. We do not have to compute these or their square roots to reject many of them. The argument is as follows. Suppose we compute one of these values in decimal and examine the least significant digit. The number cannot possibly be a perfect square if the last digit is a 2, 3, 7 or 8 as the square of no number ends with these digits. Therefore given any Y and N we can immediately reject $4/10$ of the values. This
argument rejects numbers by considering their values modulo 10. We may consider other moduli and if we use at set of p values that are mutually prime we will only have to test 1 value in 2^P. So using say 30 primes we will only have to test one value of m in 2^{30}.

In our implementation we decided to use the exact parallelism in our system (4096) and in one respect implemented the algorithm as if we had a single processor with a 4096 bit word length. We set up 30 tables corresponding to the 30 primes between 61 and 211 (the values used in the serial code). Table i corresponded to a prime p_i and has p_i entries each entry being a 4096 bit `word'. Bit m in word k it is set .true. if (Y+m+4096*k)^2-N(modulo p_i) is not potentially quadratic. These tables are computed by means of the Jacobi symbol.

These words have a repeating pattern of length p_i and each word differs from its predecessor only by its starting point within the pattern.

The loop over all values of m therefore has two computation components.

Compute the pointers to the table by the operation pointer_i = (pointer_i+4096) (modulo p_i) an operation that can be done simultaneously for all values of i.

`OR' together the selected set of masks from the tables. We can stop when all the current set of 4096 values have been rejected. We could do this by a piece of code that follows the following strategy. In this pseudo code reject and table are 4096 element logicals.

```
reject = .false.
do 10 i= 1,30
    reject= reject .or. table(pointer(i),i)
    if(all(reject)) go to 100
10  continue
```

We know however that approximately 50% of the elements in any given mask are .true. We know therefore that the probability of leaving the loop before i=15 is low.

We therefore split the loop into two phases
```
do 10 i = 1,15
    reject = reject .or. .table(pointer(i),i)
10  continue
do 20 i= 16,30
    reject = reject .or. .table(pointer(i),i)
    if(all(reject)) go to 100
10  continue
```
c arriving here if a candidate(s) has been found
c test candidates
100 start next batch

This proved very efficient and very high speed sieving and speed approaching $10^8$ tests per second are possible in Fortran and probably a factor 10 faster again if the inner loop were to be tightly coded in assembler. Many further optimizations are also possible.

The running time of any program of this type is very strongly dependant on the numbers being factorized. The program was tested on the set of 41 numbers of the form $5+\left(2^{16}+1\right)2^r$ with $r = 1 \ldots 41$.

Only two of the values needed to be factorized using the Fermat algorithm and the primality test was only applied five times.

For the complete set of numbers the total running time on a DAP 610 was 11.5 seconds, 5 seconds of which was used in the initialization phase. We were told that the original FORTRAN program ran in 276 seconds on a CRAY (unknown model) and that the initialization phase for the CRAY was also 5 seconds. The factorization phases themselves therefore ran in 6 seconds on the DAP compared with 271 seconds on the CRAY.
Monte Carlo Event Generation for LHC

Torbjörn Sjöstrand
CERN/TH, 1211 Geneva 23, Switzerland

Abstract
The necessity of event generators for LHC physics studies is illustrated, and the Monte Carlo approach is outlined. A survey is presented of existing event generators, followed by a more detailed study of the different components that appear in a complete program.

1 Introduction
A typical event at LHC energies is expected to consist of about 100 charged particles and as many neutral ones, and interesting events often have even higher multiplicities. At the nominal LHC luminosity, every bunch crossing, 15 ns apart, will produce around 20 such events. Many signals for new physics make use of collective properties of these particles, such as jets or missing transverse momentum (from neutrinos and other undetected particles). Others are based on single particles, such as electrons, muons or photons, which then have to be identified with some degree of confidence.

Almost all of the intended signals have their potential backgrounds, wherein everyday processes occasionally have fluctuations that make them look more exotic than they really are. For instance, an isolated high-momentum electron is a main signal for a number of rare processes. On the other hand, an electron produced in a $b$ quark jet is only very rarely isolated, but the total $b$ jet production rate is so huge that even a small fraction of it can be dangerous. A number of detailed questions therefore need to be answered: how well is an electron coming from the signal process really isolated?; how often do possible backgrounds give isolated electrons?; which calorimeter granularity is required to do the job?; and so on.

The task of event generators is to describe, as accurately as possible, the experimental characteristics of physics processes of interest. The main applications are as follows.

- To give physicists a feeling for the kind of events one may expect/hope to find, and at what rates.
- As a help in the planning of a new detector, so that detector performance is optimized, within other constraints, for the study of interesting physics scenarios.
- As a tool for devising the analysis strategies that should be used on real data, so that signal-to-background conditions are optimized.
- As a method for estimating detector acceptance corrections that have to be applied to raw data, in order to extract the ‘true’ physics signal.
- As a convenient framework within which to interpret the significance of observed phenomena in terms of a more fundamental underlying theory (usually the standard model).

To write a good event generator is an art, not an exact science. It is essential not to trust blindly the results of any single event generator, but always to have several cross-checks. Further, an event generator cannot be thought of as an all-powerful oracle, able to give intelligent answers to ill-posed questions; sound judgement and some understanding of the generator are necessary prerequisites for successful use. In spite of these limitations, the event generator approach is the most powerful tool at our disposal if we wish to gain
a detailed and realistic understanding of physics at the LHC before the day when real data are available.

The necessary input for event generators comes from the calculated matrix elements for the different processes, from the measured and parametrized structure functions, from models for parton showers, underlying events and fragmentation, etc., which will be discussed further below.

As the name indicates, the output of an event generator should be in the form of 'events', with the same average behaviour and the same fluctuations as real data. In generators, Monte Carlo techniques are used to select all relevant variables according to the desired probability distributions. The Monte Carlo approach ensures that the proper amount of randomness is included. Normally an 'event' is a list of all final state observable particles, i.e. hadrons, leptons, and photons, together with their momenta. The 'event' thus corresponds to what could actually be seen by an ideal detector. However, often one is only interested in the total energy and direction of a jet, rather than the detailed jet structure. Then a more crude event description, in terms of partons (≈ jets) and leptons, may be enough.

In principle, one must distinguish between an event generator and a numerical integration package for cross-sections: both can be used to evaluate the cross-section for a given process and for given cuts, but only the former gives the full multi-dimensional differential distribution of events within these cuts. In practice, this distinction is not always obvious for a large number of dedicated programs written to study one or a few specific processes: although the main application may be cross-section integration, only little additional effort is needed to generate simple 'events' which consist of a small number of outgoing partons and leptons. At the other end of the generator spectrum, there are large subroutine packages intended for general-purpose use, with many different processes included, and a full description of the production of all hadrons in an event. These packages contain many man-years of effort, 10–30 kilo-lines of Fortran code, and are generally better documented and supported than the smaller packages. Although they may not be the best for all applications, it is natural that we concentrate on them in the following.

Where does a generator fit into the overall analysis chain? In 'real life', the machine produces interactions. These events are observed by detectors, and interesting ones written to tape by the data acquisition system. Afterwards the events may be reconstructed, i.e. the electronics signals (from wire chambers, calorimeters, and all the rest) may be translated into a deduced setup of charged tracks or neutral energy depositions, in the best of worlds with full knowledge on momenta and particle species. Based on this clean-up information one may proceed with the physics analysis. In the Monte Carlo world, the role of the machine, namely to produce events, is taken by the event generators described in this report. The behaviour of the detectors – how particles produced by the event generator traverse the detector, spiral in magnetic fields, shower in calorimeters, or sneak out through cracks, etc. – is simulated in programs such as GEANT [1]. Traditionally, this latter activity is called event simulation, which is somewhat unfortunate since the same phrase could equally well be applied to what we here call event generation. A more appropriate term is detector simulation. Ideally, the output of this simulation has exactly the same format as the real data registered by the detector, and can therefore be put through the same event reconstruction and physics analysis chain, except that here we know what the 'right answer' should be, and so can see how well we are doing.

Since the full chain of detector simulation and event reconstruction is very time-consuming, often one does 'quick and dirty' studies in which these steps are skipped entirely, or at least replaced by very simplified procedures which only take into account the geometric acceptance of the detector and other trivial effects. One may then use the output of the event generator directly in the physics studies.

Programs still undergo rapid evolution: new processes are calculated and included;
improved structure function parametrizations appear; aspects of parton showering, fragmentation and decay are gradually better modelled; and even the physics landscape changes, e.g., as a function of the currently favoured value for the top quark mass. The programs that will be used at LHC are likely to look rather different from those available today. However, many of the basic principles should remain more or less unchanged.

One may also find discussions of, and comparisons between, event generators in many of the recent studies on LHC and SSC physics, such as [2, 3, 4, 5], and in the article [6]. Some of the text for the current article has been borrowed from [6, 7].

2 Monte Carlo Techniques

Quantum mechanics introduces a concept of randomness in the behaviour of physical processes. The virtue of event generators is that this randomness can be simulated by the use of Monte Carlo techniques. The authors have to use a lot of ingenuity to find the most efficient way to simulate an assumed probability distribution. A detailed description of the techniques would carry too far, but for the continued discussion some examples may be helpful.

First of all one assumes the existence of a random number generator. This is a function which, each time it is called, returns a number $R$ in the range between 0 and 1, such that the inclusive distribution of numbers $R$ is evenly distributed in the range, and such that different numbers $R$ are uncorrelated. It is not so trivial to define exactly what one means by ‘uncorrelated’, or how to ensure that the random number generator indeed produces as uncorrelated numbers as possible. Progress has been made in this area in recent years, however, and simple algorithms with very good properties are now in general use, see [8].

Let us now assume we have a function $f(x)$, with an allowed x range $x_{\text{min}} \leq x \leq x_{\text{max}}$, and with $f(x)$ non-negative in this range. We want to select an $x$ ‘at random’ such that the probability for a given $x$ is proportional to $f(x)$. One does not have to assume that the integral of $f(x)$ is normalized to unity; rather, the integral usually forms part of an overall weight factor we want to keep track of; however, that aspect is not covered further here. If it is possible to find a primitive function $F(x)$ which has a known inverse $F^{-1}(x)$, an $x$ can be found as follows (method 1):

$$
\int_{x_{\text{min}}}^{x_{\text{max}}} f(x) \, dx = R \int_{x_{\text{min}}}^{x_{\text{max}}} f(z) \, dz
\Rightarrow x = F^{-1}(F(x_{\text{min}}) + R(F(x_{\text{max}}) - F(x_{\text{min}}))).
$$

(1)

However, usually this is not the case. In an alternative method one therefore assumes that the maximum of $f(x)$ is known, $f(x) \leq f_{\text{max}}$ in the $x$ range considered. The following scheme will then yield the correct answer (method 2):

1. select an $x$ evenly in the allowed range, i.e. $x = x_{\text{min}} + R(x_{\text{max}} - x_{\text{min}})$;
2. select an $h$ evenly between 0 and $f_{\text{max}}$, i.e. $h = Rf_{\text{max}}$ (remember that, although $R$ was also used in point 1, the rules of the game are that anytime $R$ appears it means a new random number);
3. if $h \geq f(x)$ the $x$ value is rejected, and it is necessary to return to point 1 for a new try;
4. else one is done, and the most recent $x$ value is retained as final answer.

Of course, in real life there are a number of complications. On the one hand, $x$ may be a multidimensional vector with complicated boundaries for the allowed region. On the other hand, the function $f(x)$ may be rapidly varying. A common example here is a function with a singularity just outside the allowed region, such that the second method is very inefficient. Alternatively one may have an integrable singularity just at the boundary, and then the method does not work at all. In some cases it may even be difficult to know what is the appropriate $f_{\text{max}}$ to use, since the function may have several local maxima in
positions not known beforehand. Of course, the method works for any $f_{\text{max}}$ bigger than the true maximum, but if one picks $f_{\text{max}}$ unnecessarily large, one pays a price in terms of efficiency.

No single method is enough to solve all the conceivable cases. Here we just illustrate three common techniques among the many possible — more may be found e.g. in [9].

Variable transformations may be used to make a function more smooth. Thus a function $f(x)$ which blows up as $1/x$ for $x \to 0$, with an $x_{\text{min}}$ close to 0, would instead be roughly constant if transformed to the variable $y = \ln x$. Variable transformations are also often useful to simplify the shape of the boundary of the allowed $x$ region.

Special tricks can sometimes be found. Consider e.g. the generation of a Gaussian $f(x) = \exp(-x^2)$. This function is not integrable, but if we combine it with the same Gaussian distribution of a second variable $y$ it is possible to transform to polar coordinates

$$f(x) \, dx \, f(y) \, dy = \exp(-x^2 - y^2) \, dx \, dy = r \exp(-r^2) \, dr \, d\phi,$$

and now the $r$ and $\phi$ distributions may be easily generated and recombined to yield $x$ (and a second number $y$, if one so desires).

Finally, a less straightforward but very useful approach. Assume that we can find a function $g(x) = \sum_i g_i(x)$, such that $f(x) \leq g(x)$ over the $x$ range considered, and such that the functions $g_i(x)$ each are simple in the sense that we can find primitive functions and their inverses. In that case (method 3):

1. select an $i$ at random, with relative probability given by the integrals

$$\int_{x_{\text{min}}}^{x_{\text{max}}} g_i(x) \, dx = G_i(x_{\text{max}}) - G_i(x_{\text{min}});$$

2. for the $i$ selected, use method 1 to find an $x$, i.e.

$$x = G_i^{-1}(G_i(x_{\text{min}}) + R(G_i(x_{\text{max}}) - G_i(x_{\text{min}})));$$

3. select an $h = Rg(x)$;
4. if $h \geq f(x)$ reject the $x$ value and return to point 1;
5. else one is done.

For a function $f(x)$ which is known to have sharp peaks in a few different places, the generic behaviour at each peak separately may be covered by one or a few simple functions $g_i(x)$, to which one adds a few more $g_i(x)$ to cover the basic behaviour away from the peaks. By suitable selection of the relative strengths of the different $g_i$s (which may be left for the computer to figure out at an initialization stage), it is possible to find a function $g(x)$ which is matching well the general behaviour of $f(x)$, and thus to achieve a high Monte Carlo efficiency.

3 Overview of Event Generators

The perfect event generator does not exist. This reflects the limited understanding of physics in many areas. Indeed, a perfect generator can only be constructed once everything is already known, in which case experiments are superfluous. One therefore has to be satisfied with programs which are in reasonable agreement with already accumulated experience, theoretical and experimental, and which provide sensible extrapolations to higher energies. Since the ultimate goal is to look for new physics, it is also necessary to include the simulation of different alternative scenarios.

Given the complexity of the problem, the Monte Carlo approach allows a convenient division into separate subtasks. Thus, to describe an event in full, one needs to consider the following components:

1. The hard scattering matrix elements. These define the process(es) under study, and are therefore at the core of the programs.
2. The structure functions. The differential cross-sections, which are to be simulated in the programs, are given as the products of structure functions and the hard scattering matrix elements above.

3. Final state radiation. Partons in the final state may radiate. At high energies, this perturbative radiation is the dominant mechanism for building up the structure of (high-$p_t$) jets, with broad jet profiles and subjets.

4. Initial state radiation. The incoming partons may also radiate before the hard interaction, thus giving rise to additional jets close to the directions of the incoming hadrons.

5. Beam jets. Only one parton from each incoming hadron is assumed to participate in the hard interaction, and in the initial state showering. All the other partons act to produce the beam jets found along the directions of the original incoming hadrons.

6. Fragmentation and decays. Partons are not directly observable. Instead, once sufficiently removed from each other, they are fragmented into a collection of hadrons. Many of these hadrons are unstable, and subsequently decay.

Of course, this separation is very crude and schematic. Thus, one and the same $2 \rightarrow 3$ process might be described either in terms of a basic $2 \rightarrow 3$ matrix element, or in terms of a $2 \rightarrow 2$ hard scattering followed by final state radiation, or in terms of a $2 \rightarrow 2$ hard scattering preceded by initial state radiation. It is therefore important to join the different descriptions in a consistent manner, e.g. to avoid double counting.

The double counting issue is non-trivial, and in practice it has led to a split of the Monte Carlo program activity into two different approaches, which we will refer to as 'parton showers' (PS) and 'matrix elements' (ME), respectively.

In the parton shower approach, it is customary to implement only the lowest order matrix elements, i.e. as a rule, basic $2 \rightarrow 2$ processes. Initial and final state radiation are added on to the basic scattering in the shower approach proper. The showers are assumed to be universal, i.e. the shower evolution is not allowed to depend on the details of the hard scattering, but only on the gross features: energies and flavours of incoming and outgoing partons, and an overall $Q^2$ scale for the hard scattering. The approximate nature is reflected in a limited accuracy for the rate of production of additional well-separated jets, but the internal structure of jets should be well modelled. It is feasible to add fragmentation and beam jets, and thus to generate realistic representations of the events produced in hadron colliders. In this category of programs, a large fraction of the total investment is in the common shower and fragmentation routines, while the effort needed to include yet another $2 \rightarrow 2$ process is modest, if only matrix elements are known and not too complex. Some of the programs of this kind therefore allow the simulation of many different processes.

The list of such event generators is fairly small. We are aware of the following programs:

- ISAJET, by Paige and Protopopescu, current version 6.36 [10].
- PYTHIA, by Bengtsson and Sjöstrand, current version 5.5 [11].
- HERWIG, by Marchesini and Webber, current version 5.3 [12].
- COJETS, by Odorico, current version 6.11 [13].
- DTUJET, by Ranft et al. [14].
- FIELDJET, by Field et al. [15].
- The Fire-String program by Angelini et al. [16].
- FRITIOF, by Andersson et al., current version 6.0 [17].

Without passing judgement on quality, the ordering above does reflect an element of quantity: ISAJET and PYTHIA are clearly more versatile than the others, with HERWIG up-and-coming, while the latter four programs only cover QCD jets and minimum bias events.
The matrix element approach is represented by another class of programs. Here the emphasis is on the use of exact higher-order matrix elements. The analytic formulae in the programs are considerably more complicated, and the phase space generation machinery more advanced. The big investment here is in the matrix element calculation itself — usually these programs are written by the same people who calculated the matrix elements in the first place — and in selecting the kinematic variables in an efficient way. There is therefore less impetus for a common approach to many disparate processes. Since the precision aspect is important, it is not feasible to attach a simple, generic parton shower picture. Normally, therefore, only a fixed (small) number of partons are generated. Since most modern fragmentation models are tuned to be attached at the end of the parton shower evolution, fragmentation and beam jet treatments also become less interesting. These programs therefore mainly generate parton configurations of 'pencil jets', rather than events as they may appear in a detector.

The number of matrix element programs is considerably higher than the number of parton shower programs: once a matrix element has been calculated, the Monte Carlo approach is usually the most convenient way to obtain physical cross-sections. Therefore many calculations are directly turned into programs. It is not possible in this report to give a complete list of all programs of this kind, some of which are publicly maintained and others which are not. Two programs contain matrix elements for widely different purposes:

- PAPAGENO, by Hinchliffe [18].
- EUROJET, by van Eijk et al. [19].

A few others will be mentioned in connection with the processes they simulate.

The parton shower and matrix element programs fill somewhat complementary functions. The former are convenient for exploratory work: it is fairly easy to simulate a new, postulated physics process in sufficient detail to establish experimental feasibility, and to try out the tools needed to separate signal from background. For high-precision measurements of an established process, on the other hand, one needs the higher order matrix elements. The matrix element programs are also more convenient for generating events within very specific phase space regions, since the cuts can be included from the start. With parton shower based programs it is necessary to generate more inclusive event samples and afterwards discard those events that do not fulfil the requirements, a procedure which can often be very inefficient.

3.1 Kinematics and cross-sections

In this section we describe how the hard scattering process is generated. The example is for the case of a $2 \rightarrow 2$ process, with mass effects neglected, but the same basic principles apply also for other cases.

Consider two incoming protons in their CM frame, each with energy $E_{\text{beam}}$. The total CM energy-squared is then $s = 4E_{\text{beam}}^2$. The two partons that enter the hard interaction do not carry the total beam momentum, but only fractions $x_1$ and $x_2$, respectively, i.e. they have four-momenta

$$
\begin{align*}
    p_1 &= E_{\text{beam}}(x_1; 0, 0, x_1), \\
    p_2 &= E_{\text{beam}}(x_2; 0, 0, -x_2).
\end{align*}
$$

(5)

The invariant mass-squared of the two partons is defined as

$$
\hat{s} = (p_1 + p_2)^2 = x_1 x_2 \hat{s}.
$$

(6)

Instead of $x_1$ and $x_2$, it is often customary to use $\tau$ and either $y$ or $x_F$:

$$
\tau = x_1 x_2 = \frac{\hat{s}}{s}.
$$

(7)
\[ y = \frac{1}{2} \ln \frac{x_1}{x_2}; \]
\[ x_F = x_1 - x_2. \]

In addition to \( x_1 \) and \( x_2 \), two additional variables are needed to describe the kinematics of a scattering \( 1 + 2 \rightarrow 3 + 4 \). One corresponds to the azimuthal angle \( \phi \) of the scattering plane around the pp axis. This angle is isotropically distributed (unless the protons are polarized), and so need not be considered further. The other variable could have been picked as \( \hat{\theta} \), the polar angle of parton 3 in the CM frame of the hard scattering. However, the conventional choice is to use the variable

\[ \hat{t} = (p_1 - p_3)^2 = (p_2 - p_4)^2 = -\frac{s}{2} (1 - \cos \hat{\theta}), \]

with \( \hat{\theta} \) defined as above. It is also customary to define \( \hat{u} \),

\[ \hat{u} = (p_1 - p_4)^2 = (p_2 - p_3)^2 = -\frac{s}{2} (1 + \cos \hat{\theta}), \]

but \( \hat{u} \) is not an independent variable since

\[ \hat{s} + \hat{t} + \hat{u} = 0. \]

The cross-section for the process \( 1 + 2 \rightarrow 3 + 4 \) may be written as

\[
\sigma = \int \int \int dx_1 \ dx_2 \ dt \ f_1(x_1, Q^2) f_2(x_2, Q^2) \frac{d\sigma}{dt} = \int \int \frac{d\tau}{\tau} \ dy \ dt \ x_1 f_1(x_1, Q^2) x_2 f_2(x_2, Q^2) \frac{d\sigma}{dt}. \]

The \( f_i(x, Q^2) \) are the structure functions of the proton, which express the probability to find a parton of species \( i \) with a momentum fraction \( x \) inside a proton probed at a virtuality scale \( Q^2 \). The choice of \( Q^2 \) scale is ambiguous; one common alternative is the transverse momentum-squared

\[ Q^2 = p_{1\perp}^2 = \frac{s}{4} \sin^2 \hat{\theta} = \frac{\hat{t} \hat{u}}{s}. \]

The \( \frac{d\sigma}{dt} \) expresses the differential cross-section for a scattering, as a function of the kinematical quantities \( \hat{s}, \hat{t} \) and \( \hat{u} \). It is in this function that the physics of a given process resides.

The performance of a machine is measured in terms of its luminosity \( \mathcal{L} \), which is directly proportional to the number of particles in each bunch and to the bunch crossing frequency, and inversely proportional to the area of the bunches at the collision point. For a process with a \( \sigma \) as given by eq. \((13)\), the differential event rate is given by \( \sigma \mathcal{L} \), and the number of events collected over a given period of time

\[ N = \sigma \int \mathcal{L} \ dt. \]

### 3.2 Hard scattering subprocesses

Lists of subprocesses included in Monte Carlos are found in Tables 1 and 2. These tables should be read as follows. For ISAJET, PYTHIA and PAPAGENO, a ‘•’ indicates that the process is included and a ‘−’ that it is not. In the column ‘other PS’ (PS = parton
Table 1: Standard model physics processes included in the event generators studied. See text for program notation. ‘f’ stands for fermion, ‘V’ for W or Z, and ‘Q’ for heavy quark.

<table>
<thead>
<tr>
<th>Process</th>
<th>ISAJET</th>
<th>PYTHIA</th>
<th>other PS</th>
<th>PAPAGENO</th>
<th>other ME</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QCD</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QCD jets</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>E, NJ</td>
</tr>
<tr>
<td>$qar{q}, gg \rightarrow t\bar{t}$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>E</td>
</tr>
<tr>
<td>$q^b \rightarrow q't$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td>E</td>
</tr>
<tr>
<td>$J/\Psi/Y$ (direct)</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>E</td>
</tr>
<tr>
<td>minimum bias</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>diffractive</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>elastic</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Prompt photons</strong></td>
<td></td>
<td></td>
<td>H</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>$gg \rightarrow \gamma, q\bar{q} \rightarrow \gamma$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>$q\bar{q} \rightarrow \gamma\gamma$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$gg \rightarrow \gamma\gamma$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>W/Z production</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q\bar{q} \rightarrow V$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>LD</td>
</tr>
<tr>
<td>$gg, q\bar{q} \rightarrow V_q(g, g)$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td>LD</td>
</tr>
<tr>
<td>$q\bar{q} \rightarrow VV, V\gamma$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>BZ, BH</td>
</tr>
<tr>
<td>$q\bar{q}, gg \rightarrow VV_q(g, g)$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>VV, BH</td>
</tr>
<tr>
<td>$gg \rightarrow VV, V\gamma$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>$q\bar{q} \rightarrow V^<em>V^</em>$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>GG</td>
</tr>
<tr>
<td>$gg \rightarrow ZQQ$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>LD</td>
</tr>
<tr>
<td>**Standard model $H^0 (m_H \leq 800$ GeV)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q\bar{q} \rightarrow H^0$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>GG</td>
</tr>
<tr>
<td>$gg \rightarrow H^0$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td>BG</td>
</tr>
<tr>
<td>$VV \rightarrow H^0$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td>BG</td>
</tr>
<tr>
<td>$q\bar{q} \rightarrow VH^0$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>$gg, qg, q\bar{q} \rightarrow H^0_q(g, g)$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td>HV</td>
</tr>
<tr>
<td>$H^0 \rightarrow VV$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td>BG, HV</td>
</tr>
<tr>
<td>$H^0 \rightarrow V^<em>V^</em>$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>$H^0 \rightarrow f\bar{f}$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>$H^0 \rightarrow gg$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>$H^0 \rightarrow \gamma\gamma$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>$H^0 \rightarrow \gamma Z^0$</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>**Standard model $H^0 (m_H \geq 700$ GeV)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$VV \rightarrow VV$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td>BG</td>
</tr>
<tr>
<td>$gg \rightarrow VV$</td>
<td>•</td>
<td>•</td>
<td>H</td>
<td>•</td>
<td>GG</td>
</tr>
</tbody>
</table>
Table 2: Non-standard model physics processes included in the event generators studied. See text for program notation. In addition to notation for Table 1, \( V' \) stands for \( W' \) or \( Z' \), \( R \) for a horizontal boson, and \( L \) for heavy lepton.

<table>
<thead>
<tr>
<th>Process</th>
<th>ISAJET</th>
<th>PYTHIA</th>
<th>other PS</th>
<th>PAPAGENO</th>
<th>other ME</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-standard Higgs particles</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h, H, A \text{ as above} )</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( Z^* \rightarrow h^0 A^0, H^0 A^0 )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( q\bar{q} \rightarrow H^+ )</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( gb \rightarrow H^- t )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( \gamma^<em>/Z^</em> \rightarrow H^+ H^- )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( t \rightarrow H^+ b )</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( H^+ \rightarrow f\bar{f} )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Supersymmetry</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q\bar{q}, gg \rightarrow \tilde{q}\tilde{q} )</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>UA, BT</td>
</tr>
<tr>
<td>( q\bar{q}, gg \rightarrow \tilde{g}\tilde{g} )</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>UA, BT</td>
</tr>
<tr>
<td>( qg \rightarrow \tilde{q}\tilde{g} )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>UA, BT</td>
</tr>
<tr>
<td>( q\bar{q} \rightarrow \tilde{g}\tilde{V} )</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>UA</td>
</tr>
<tr>
<td>( gg \rightarrow \tilde{g}\tilde{V} )</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>UA</td>
</tr>
<tr>
<td>( \tilde{q}, \tilde{g}, \tilde{V} \text{ decays} )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>UA, BT</td>
</tr>
<tr>
<td><strong>New Gauge Groups</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q\bar{q} \rightarrow V' )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( VV \rightarrow V' )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( V' \rightarrow f\bar{f} )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( V' \rightarrow VV )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( q\bar{q}' \rightarrow R \rightarrow q''q''' )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Fourth Generation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q\bar{q}, gg \rightarrow Q\bar{Q} )</td>
<td>*</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>( V/V' \rightarrow Q\bar{Q}, L\bar{L} )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( q\bar{q}' \rightarrow q''Q )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Other Topics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>contact interactions</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>axigluons</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>leptquarks</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>strongly interacting ( V )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>( q^* \text{ (excited fermions) } )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
shower programs) a 'o' indicates this is something found in most or all programs in this
category, while an 'H' appears if only HERWIG includes it and a blank if no program does.
In the column 'other ME' (ME = matrix element programs), an 'E' indicates a process
included in EUROJET, and other letters indicate processes found in other programs, as
explained further in the process-specific descriptions below.

The tables should be taken as indicative only, since there is a continuous evolution
of many programs. Furthermore, one and the same process may be treated differently
in different programs. Below we will give some comments on a few of the processes, to
illustrate the degrees of freedom open to Monte Carlo authors.

3.2.1 QCD

Exact Born term cross-sections, for up to five jets in the final state, are available
in the NJETS program of Kuijf and Berends ('NJ' of Table 1), see [20], which is the most
advanced in this category. This program also contains approximate expressions for up to
eight jets.

Complete loop calculations have been performed up to \( O(\alpha_s^3) \). These are imple-
mented in the numerical integration programs of two groups [21], but no event generators
exist so far.

Most programs only contain the lowest order Born term cross-sections for heavy
flavour production. For top this may be sufficient, i.e. higher order contributions effectively
contribute an overall K factor, but do not significantly change the production character-
istics of top. However, at LHC energies, it is not correct to use only the Born term to
estimate \( b \) or \( c \) production, since these quarks receive major higher order contributions,
both by flavour excitation and by parton shower evolutions.

Minimum bias physics is discussed together with beam jets below.

Even when diffractive and elastic scattering is included in programs, the treatment
is fairly primitive, and likely to be insufficient for LHC physics. Several major features
are missing, like high-\( p_T \) jet production in diffractive events [22].

3.2.2 Prompt photons

Complete next-to-leading order programs for prompt photon production are avail-
able from two groups [23], but both are intended for cross-section calculation rather than
event generation. Leading order formulae are contained in many event generators. Some
parton shower algorithms also include the emission of photons as part of the evolution.

The \( gg \to \gamma \gamma \) graph contains a quark box. The cross-section is reasonably compact
in the limit of vanishing quark mass, but very complex if the correct quark mass depend-
dence is included. Therefore often the massless formulae are used, with the number of
flavours suitably chosen. PYTHIA contains the full formulae as an option, but these then
are numerically unstable in some regions of phase space, and therefore not easy to use.

3.2.3 \( W/Z \) production

The most complete \( W/Z \) ME program is the 'Leiden-Durham \( W'/VECBO \)S pro-
gram ('LD' of Table 1), which contains the production of a \( V \), i.e. a \( W \) or a \( Z \), plus 0, 1,
2, 3 or 4 jets, see [24]. No loop corrections are available in this program, but analytical
formulae exist up to second order in \( \alpha_s \) [25]. Programs for the production of \( V + V \) and
\( V + \gamma \) are also available, in [26] ('BZ' of Table 1) with special emphasis on the possibility
of testing for anomalous couplings in triple gauge boson vertices. The production of a \( VV \)
pair plus one additional jet is found in two programs: in VVJET [27] ('VV' of Table 1)
and in [28] ('BH' of Table 1); the latter also contains matrix elements for a \( VV \) pair plus
two jets. In all the programs above, subsequent \( V \) decays are included, with full angular
correlations.

As in \( \gamma \gamma \) pair production, \( VV \) pairs may also be produced from a \( gg \) initial state,
via a quark box. The rates may be sizeable, thanks to the large value of the gluon structure
functions at the small $x$ values probed by LHC, and interference with the Higgs signal is of particular importance for Higgs searches. The program GGZZ simulates this process [29] ('GG' of Table 1).

The parton shower programs tend to give a fairly good description of $V$ production at current energies. However, the rate of high-$p_T$ $V$ production is not so well reproduced if the starting point is the $q\bar{q} \to V$ matrix element. One may instead use the $gg \to Vg$ and $qq' \to Vq$ matrix elements, in which case at least one high-$p_T$ jet is assured from the start, and then include showering to generate additional jets. This gives a better description at high $p_{T,V}$, but cannot be used to describe inclusive $V$ production, since the $2 \to 2$ matrix elements are divergent for $p_T \to 0$. The choice between the two descriptions therefore has to depend on the application. In ISAJET a special option is available, in which the $2 \to 2$ matrix elements have been regularized (by hand) in the limit $p_{T,V} \to 0$, and so a good description is obtainable over the whole $p_T$ spectrum.

For intermediate mass Higgs background studies, the $Z+(Z^*/\gamma^*)$ (where * denotes that the interesting configurations are those with the particle far off mass-shell) and $Zbb$ channels are of particular interest. The latter process is calculated in [30], and is now included in a few generators, although still with an inefficient selection of phase-space points.

### 3.2.4 Standard model $H^0$

A single unified description of Higgs production and decay characteristics, valid for all Higgs masses, would be very complex. In practice, two different descriptions are in use in programs. For a reasonably light Higgs, and thereby a reasonably narrow one, the 'signal' and the 'background' graphs do not interfere significantly, so that it is possible to separate the process into Higgs production and Higgs decay. If the Higgs is heavy, this is no longer possible but, in this region, mainly the $VV \to H \to VV$ graphs are of experimental interest, and so only full interference with the $VV \to VV$ background need be included.

A light or intermediate mass Higgs is predominantly produced by $gg \to H$. The process $VV \to H$, i.e. properly $qq' \to q''q'''H$, also contributes. This process is included with the full matrix elements in PYTHIA, HERWIG and PAPAGENO. In ISAJET the effective $W$ approximation is used, which is known to be good for $m_H \gg m_W$, but not so good for lower Higgs masses.

In the description of Higgs decays, two new aspects have played a particular role in recent activity. One is the introduction of running quark masses for couplings $H \to q\bar{q}$; this typically leads to a reduction of the quark partial widths by a factor of around 2. At intermediate Higgs masses, where the $H \to b\bar{b}$ decays dominate, some other branching ratios are enhanced by the same factor 2, notably $H \to \gamma\gamma$. Running quark masses are included in PYTHIA and HERWIG, but not in ISAJET. The other new aspect is $H \to V^*V^*$ decays, i.e. where one or both final state gauge bosons are significantly off mass-shell. Particularly interesting are the $H \to ZZ^* \to 4\ell$ decays, which now are found in ISAJET, PYTHIA and HERWIG.

For the heavy Higgs scenario, both ISAJET and PYTHIA rely fully on the effective $W$ approximation for $VV \to VV$ matrix elements. In both programs the incoming $V$ bosons are assumed longitudinally polarized, as are the outgoing in PYTHIA, while ISAJET includes all polarization combinations in the final state. A more detailed description, based on exact matrix elements with full interference between all graphs that can yield $VV$ plus two jets in the final state, is found in [31] ('BG' of Table 1); full angular correlations in the $V$ decays are also included.

Finally, just as for the description of high-$p_T$ $V$ production, it may be convenient to have a description of a $H$ recoiling against a jet; this is available in the program HVVJET [32] ('HV' of Table 1).
3.2.5 Non-standard Higgs particles

Little effort has gone into scenarios with more Higgses than in the standard model. However, recently many of the production processes in the two-Higgs-doublet scenario of the Minimal Supersymmetric Standard Model (MSSM) were included in PYTHIA, and steps in the same direction have been taken in HERWIG. In the MSSM, there are five physical Higgses: two neutral scalars $h^0$ and $H^0$, one neutral pseudoscalar $A^0$, and two charged particles $H^\pm$. The production of the neutral particles follows the same pattern as that of the standard model $H^0$, except that couplings are changed, and that pair production e.g. of $h^0 + A^0$ become of interest. It also becomes necessary to cover the possibility of sequential decays of one Higgs state into another, e.g. by $Z^0$ emission. The charged Higgs may be produced singly or in pairs; one potentially significant source is top decays.

3.2.6 Supersymmetry

SUSY is an important area to be explored at LHC. Several different particles should be searched for, in particular squarks, gluinos and a host of gauginos. Two main programs in this area are ISAJET and UA2SUSY. As the name indicates, the latter ('UA' of Table 2) is an upgrade of a dedicated program written inside the UA2 collaboration [33]. A further program is found in [34] ('BT' of Table 2). Recent developments include a special emphasis on a flexible and detailed modelling of all sequential decay chains predicted for different parameter sets of the MSSM.

3.2.7 New gauge groups

A number of different scenarios can give rise to new gauge particles, here denoted $V'$ ($= Z'^0$ or $W'^\pm$). In PYTHIA, vector and axial couplings of fermions to the $V'$ have been left as free parameters; it is therefore possible to simulate most of the alternatives on the market by judicious choices. Couplings of a $V'$ to the standard model gauge bosons can show a richer structure, and only a few of the possibilities are available here.

A specific model for a horizontal boson $R$, i.e. a boson which couples to generation number, has been included as a separate alternative in PYTHIA.

3.2.8 Fourth generation

With the current LEP limits on the number of light neutrino species, the prospects are slim for a standard fourth generation of fermions. Should there still be some interest in heavy standard quarks or leptons, the event generators are available, since only trivial extensions of the standard description of top are involved.

3.2.9 Other topics

The list of possible extensions to, or deviations from, the standard model is long, and only a few are found in Table 2. Among the most interesting ones are the prospects of a strongly interacting $V$ sector, as could arise if the standard model Higgs were absent or, at least, much heavier than the 1 TeV mass scale directly probed. Some of the scenarios proposed in the literature have been implemented in PYTHIA.

3.3 Structure Functions

The proton is not a static object. In addition to the three valence quarks, virtual gluons and quark-antiquark pairs are continually created and annihilated. Currently a first-principles complete picture does not exist — maybe lattice QCD studies one day will provide that. Meanwhile, one makes do with a simple probabilistic picture, where the structure functions $f_i(x, Q^2)$ give the probability to find a parton $i$ with fraction $x$ of the proton momentum, if the proton is probed at a scale $Q^2$. As the scale is increased,
more and more short-lived fluctuations can be probed. Therefore the structure functions change in a characteristic fashion expressed by the evolution equations

$$\frac{\partial f_i(x, Q^2)}{\partial \ln Q^2} = \sum_j \int_x^1 \frac{dz'}{z'} f_j(z', Q^2) \frac{\alpha_S(Q^2)}{2\pi} P_{j\rightarrow i}\left(\frac{z}{z'} \right),$$

(16)

where $\alpha_S$ is the strong coupling constant. The splitting kernels $P_{j\rightarrow i}$ are described further below, eq. (18). While thus the $Q^2$ dependence is predicted by theory, it is necessary to determine the functions at some fixed scale $Q_0^2$ by comparison with data, in particular from deep inelastic scattering experiments.

A community of people is involved in the analysis of data and the extrapolation to unmeasured regions, based on the QCD evolution equations. The end result of these efforts is new structure function sets, with some region of validity in the $(x, Q^2)$ plane. In the past, the number of sets available was fairly limited; for applications at the large $Q^2$ scales of LHC/SSC, only the EHLQ parametrizations [35] could be used, which is why these are still found as defaults in many programs.

More recently, the pace has picked up, and now new sets appear almost monthly. A review of, and comparison between, most of these is found in [36]. One conclusion is that many of the older sets do not do well when compared with current data, and therefore should no longer be used. Also some of the newer sets perform less well. In part, this is deliberate: given the large uncertainties involved, most authors do not provide a single 'best' set, but rather prefer to produce many different sets, which together are supposed to bracket the 'right' answer. The differences between these sets come from the correlation between the choice of $\Lambda$ value (in $\alpha_S$) and the choice of gluon structure function, from different assumptions about the behaviour of structure functions at low $z$, from different choices of strange quark distributions at low $Q^2$, etc.

Since all sets of structure functions are limited in validity to given $x$ and $Q^2$ ranges (in particular, $x > 10^{-6}$ to $10^{-4}$, depending on the set), their use for applications at LHC/SSC energies need extra care. Cross-sections and differential distributions (e.g. for $b$ or $c$ quark production) could be affected. To overcome the problem in part, Monte Carlo authors may have to introduce further assumptions themselves.

An additional element of disparity comes from the choice of order and renormalization scheme. The three main alternatives are leading order, next-to-leading order in the $\overline{MS}$ scheme, and next-to-leading order in the DIS scheme. For high precision measurements, it is essential to use the same conventions for matrix elements and structure functions, and here probably little confusion exists. The appropriate choice to use for parton shower based programs may be less clear — while basically leading log, these programs do include some next-to-leading log contributions.

A main programming issue for structure functions is whether to use grids or parametrizations. In the former approach, the output of the evolution programs is stored directly as grids in the $(x, Q^2)$ plane, and desired values can be obtained by interpolation in these grids. The drawback is that thousands of real numbers have to be transferred to each new computer as external files, which makes programs a little less easily transportable. The advantage is that interpolation is usually fast. In the parametrization approach, smooth functions are fitted to the grid values, and subsequent use is based on these fits. This way the number of real values that characterize a structure function set is significantly reduced, hopefully without any loss of information — the most spectacular example is the very compact parametrizations by Morfin and Tung [37]. Such parametrizations can easily be included in the code of an event generator, and thus there are no transport problems. Since the evaluation typically involves logarithms and exponents, it may be significantly slower than in the grid interpolation approach, however.

Recently, a program PDFLIB was released [38], which puts together basically all existing structure functions into one single package, with a common calling structure. This
has greatly simplified the task of Monte Carlo authors, who now only have to provide an interface to this library.

3.4 Initial and Final State Showers

In the parton shower approach, a hard $2 \rightarrow 2$ scattering is convoluted with initial and final state radiation to build up multiparton final states. Of the two showering types, final state radiation is theoretically and experimentally well understood, while initial state radiation remains less well understood.

As noted above, the shower approach is expected to do a good job for small-angle emission, which is the one that determines the internal structure of jets. It is inferior to the matrix element approach for the rate of well-separated jets but can, to some extent, be tuned to give a reasonable overall description also in this region.

In leading log, the probability $P$ for branchings $q \rightarrow qg$, $g \rightarrow gg$, and $g \rightarrow q\overline{q}$ is described by the standard evolution equations

$$\frac{dP}{d\ln Q^2} = \int dz \frac{\alpha_s(Q^2)}{2\pi} P_{a \rightarrow bc}(z),$$

where

$$P_{q \rightarrow qg}(z) = \frac{4}{3} \frac{1 + z^2}{1 - z},$$

$$P_{g \rightarrow gg}(z) = 3 \frac{(1 - z(1 - z))^2}{z(1 - z)},$$

$$P_{g \rightarrow q\overline{q}}(z) = \frac{1}{2} (z^2 + (1 - z)^2).$$

The $z$ variable describes the sharing of energy (and momentum) between the daughter partons, with parton $b$ taking a fraction $z$ and $c$ the remaining $1 - z$ of the original $a$ energy. The probability for soft gluon emission is divergent, and is normally regularized by requiring some minimum parton energy, i.e. $z_{\text{min}} \leq z \leq z_{\text{max}}$.

Equation (17) describes the probability for a single branching $a \rightarrow bc$. Once formed, the daughters $b$ and $c$ may in their turn branch, and so on, so that a tree-like structure develops. The shower evolution is cut off at some minimal scale, typically $Q_0 \approx 1$ GeV. Below this scale, perturbation theory is assumed to break down, and non-perturbative fragmentation takes over.

Although eq. (17) does not seem to distinguish between initial and final state radiation, in fact the difference is quite significant. This will be outlined in the following.

Final state showers are timelike: the two outgoing partons of a $2 \rightarrow 2$ scattering each has $m^2 = E^2 - p^2 \geq 0$. The evolution is therefore in terms e.g. of $Q^2 = m^2$, and in each successive branching the daughters are constrained by kinematics to have a smaller $m^2$ than that of their mother.

The naive leading log parton shower picture is modified by coherence effects, which can be taken into account by the inclusion of angular ordering [39], i.e. not only are virtualities successively degraded, but so are the opening angles of branchings. The scale of $\alpha_s$ is also changed from $m^2$ to $p^2_\perp \approx z(1 - z)m^2$. Further details on the theory of timelike showers may be found in several reviews, e.g. [40, 41].

On the experimental front, final state showers have been much studied in $e^+e^-$ annihilation; since no initial state QCD showers appear in $e^+e^-$, and since the production graph is $s$-channel only, the analysis is simpler than in hadron collisions. The recent LEP results underline how well existing showering programs do, see e.g. [42, 43]. It is seldom that disagreements between data and programs like JETSET (which is the program used for showering in PYTHIA) or HERWIG reach the 10% level. Even more importantly,
with parameters tuned at LEP, programs also do a good job of describing data at lower energies, at PEP, PETRA and TRISTAN. Confidence in extrapolations to higher energies is therefore high.

Anytime one has to consider the hadronic decay of a colour singlet particle in hadron colliders, such as \( W, Z, H \), etc., the \( e^+e^- \) experience is directly applicable, and predictive power high. In principle, questions could be raised whether colour exchange might take place between the partons of the decaying singlet particle and the partons of the underlying event; such effects could modify event topologies, but probably not drastically. When the hard process does not go through a colour singlet intermediate state, on the other hand, there are significant ambiguities in how to begin the shower evolution at high virtualities, such that the proper amount of multijet activity is obtained. Once a choice is made here, the subsequent evolution is again well under control.

Initial state radiation is considerably more difficult to model. The shower is initiated by a parton selected from structure functions at small \( Q^2 \). This parton may now branch, but in the branching only one daughter is timelike, whereas the other is spacelike, i.e. \( m^2 < 0 \). The timelike parton may develop a shower, very much like the final state radiation case, although typically with less allowed phase space and therefore less extensive. The spacelike parton may branch once again, to a new pair of one timelike and one spacelike daughter, etc. The sequence of spacelike daughters is terminated at the hard interaction: a hard \( 2 \to 2 \) (QCD) process consists of two incoming spacelike partons and two outgoing timelike ones. In leading log language, the virtuality \( Q^2 = -m^2 \) of the sequence of spacelike partons is required to increase monotonically, and is constrained from above by the \( Q^2 \) scale of the hard interaction.

The inclusive parton distribution at each \( Q^2 \) scale is given by structure functions evolved according to eq. (16) — the difference between eq. (16) and eq. (17) is that the latter refers to the branchings of an individual parton while the former considers the evolution of the parton density as a whole. In a Monte Carlo, it is therefore not necessary to perform the initial state evolution before the hard interaction is selected. Rather, one may use already evolved structure functions to select the kinematics of the hard interaction, and only thereafter reconstruct the shower history that preceded this interaction. This is conveniently done in terms of the ‘backwards evolution’ scheme [44], where the reconstruction of the spacelike evolution is begun at the hard interaction and thereafter gradually carried to earlier and earlier times, until the cut-off scale \( Q_0 \) is reached.

In recent years, theoretical progress has been made in including coherence corrections to the leading log picture [45]. The complexity of these corrections is such that no program includes all effects in full, however. HERWIG is the program that contains the most advanced machinery. It has still not been clarified exactly how big the differences are compared to the more simplminded approaches in other programs.

### 3.5 Beam Jets

The description of beam jets, i.e. the physics of underlying events and minimum bias events, remains the least well understood aspect of Monte Carlo modelling of hadronic events. It is therefore possible to choose many possible approaches.

The most naïve would be to associate each beam remnant with a single jet, which typically would have to contain a leading baryon, but for the rest look like an ordinary quark jet. This approach does not work, in that it gives too low an average charged multiplicity and too narrow a multiplicity distribution compared to data.

One simple way out is to decouple the fragmentation of beam jets from that of ordinary jets. This is done e.g. in COJETS and HERWIG, where the particle multiplicity is selected according to a parametrization of \( pp \) data, and particles thereafter distributed according to longitudinal phase space, i.e. uniformly in rapidity. The underlying physics is here left unexplained.

An explanation from first principles may be obtained if the composite structure of
the proton is invoked, to motivate the possibility that several parton-parton interactions may take place in one and the same proton-proton collision. In ISAJET and DTUJET this is achieved within the Cut Pomeron/Dual Topological Unitarization framework, which uses unitarity arguments to derive a multiplicity distribution in the number of parton-parton interactions per event. Longitudinal momentum sharing between beam partons is based on theoretical low-$Q^2$ structure functions, while partons do not have any transverse momenta. In PYTHIA the rate of multiple parton-parton interactions is based on the standard perturbative QCD formulae, with an effective cut-off at around $p_{\perp \text{min}} \approx 1.6$ GeV, a number that does not come out of any physics analysis but is obtained by a tuning to data. Here partons thus have both longitudinal and transverse momenta.

If, in the end, the programs above agree reasonably well, it is mainly because they have been tuned to the same data, making use of the freedom inherent in all current approaches. However, detailed comparisons between programs and data are scarce.

Theoretical work on the structure of minimum bias events has been carried out in particular by Levin and Ryskin [46]. Their approach is also based on a multiple parton-parton interaction scenario. Compared to the models above, particular emphasis is put on saturation effects at small $x$. Saturation can arise when the local density of partons becomes so large that not only parton branchings but also parton recombinations have to be taken into account. This saturation is predicted to set in sooner than given by naive estimates, since a large fraction of the partons inside a proton are assumed to be concentrated in a few 'hot spots'. If correct, naive extrapolations to LHC energies, as embodied in current event generators, may fail. Some first hints on the validity of the Levin-Ryskin model may come already with HERA.

3.6 Fragmentation and Decays

Fragmentation is a non-perturbative phenomenon, and as such is not yet understood from first principles. As with timelike parton showers, experience from $e^+e^-$ annihilation helps constrain models significantly [47, 42, 43]. Three different main fragmentation schools exist: string (found in PYTHIA and FRITIOF), cluster (HERWIG) and independent (e.g. ISAJET, COJETS and EUROJET) fragmentation.

In the string and cluster approaches, the colour topology of the event affects the distribution of the final state hadrons. If two partons share a colour-anticolour pair, a string (or set of clusters), is stretched between these partons, such that low-momentum hadrons are produced predominantly in that angular range. In independent fragmentation, each parton fragments on its own, such that soft particles are evenly distributed in azimuth around the respective jet axis. Hard particles do not distinguish between models, but are always produced close to the original parton directions.

The string and cluster models are known to give good agreement with $e^+e^-$ data over a wide range of energies, and are expected to work well also at higher energies. The independent fragmentation model currently is not much used in $e^+e^-$, and is known to be unable to describe some critical distributions, although many other distributions can again be described well. On theoretical grounds, independent fragmentation is disfavoured due to a number of (conceptually) ugly features. Differences between models are difficult to find in hadron collisions, because here the physics is considerably more smeared out by a variety of effects. However, predictions exist and may one day be tested with sufficient precision. These predictions involve both the colour flow internally between three or more high-$p_\perp$ partons, and between the high-$p_\perp$ partons and the beam remnants.

It should be recognized that not all aspects are fully understood. Consider e.g. $qg \rightarrow qg$. In this process there are two possible colour flow topologies, and therefore two possible string configurations. However, the standard QCD matrix elements in fact also contain an interference term between the two, where the colour flow is not well defined. For practical applications, this term is small and may normally be neglected but, ideologically, it is not known how to achieve a fully correct description.
A majority of the particles produced in the fragmentation step are unstable and decay further. Almost all programs therefore include decay routines, more or less similar to each other. Decay data are taken from [48], where available, and according to the best understanding of the program author, where not. There are some differences in level of sophistication, with respect to inclusion of decay matrix elements and polarization information, but seldom does this give readily visible experimental consequences.

4 Programming Issues
4.1 How to Run a Program

Each program has its own style of usage. Some require a user-written main program where variables are set and the program routines called, in others the main program comes with the package and it calls on user-provided subroutines at specific moments, in others still the main program of the package reads a deck of data cards for instructions. The possibilities for the user to modify the basic behaviour are endless and often without common structure, not only between programs but also inside one and the same program. It is not the intention to go through all of that here. Instead we give a general overview, and then provide an explicit example.

Generically, the usage of an event generator takes place in three steps.

1. Initialization step. It is here that all the basic characteristics of the coming generation are specified. This includes, among others, the following points.
   - To select which processes shall be generated.
   - To set constraints on the kinematics and the flavour characteristics of the process.
   - To define the physics scenario, e.g. masses of unknown particles.
   - To pick structure functions and other such aspects of the generation.
   - To switch off generator parts not needed, or in other ways to modify the coming generation.
   - To initialize the generator, e.g. to let it find maxima of differential cross-sections.
   - To book histograms, reset user counters, etc.

2. Generation loop. This is the main part of the run, and includes the following tasks.
   - To generate one event at a time.
   - To print a few of the events, to be able to check that everything is working as intended or to check up on anomalies.
   - To analyze events for the properties of interest.
   - To add the results of the analysis to histograms etc.
   - To write events to tape, and/or feed them to a detector program.

3. Finishing step. Here the tasks are as follows.
   - To print deduced cross-sections and other summary information from the generator.
   - To print histograms and other user-specified summary information.

To illustrate this structure, imagine a toy example, where one wants to simulate the production of a 300 GeV Higgs particle. In PYTHIA, a program for this might look something like the following.

```
C...Common blocks.
COMMON/LUJETS/N,K(4000,5),P(4000,5),V(4000,5)
COMMON/LUDAT1/MSTJ(200),PARU(200),MSTJ(200),PARJ(200)
COMMON/LUDAT2/KCHG(500,3),PMA5(500,4),PARF(2000),VCKM(4,4)
COMMON/PYSUBS/MSLE,MSUB(200),KFIN(2,-40:40),CKIN(200)
COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
COMMON/PAWC/HBOOK(10000)
```
C...Number of events to generate; switch on proper processes.
   NEV=1000
   MSEL=0
   MSUB(102)=1
   MSUB(123)=1
   MSUB(124)=1

C...Select masses and kinematics cuts.
   PMAS(6,1)=140.
   PMAS(25,1)=300.
   CKIN(1)=290.
   CKIN(2)=310.

C...For simulation of hard process only: cut out unnecessary tasks.
   MSTP(61)=0
   MSTP(71)=0
   MSTP(81)=0
   MSTP(111)=0

C...Initialize and list partial widths.
   CALL PYINIT('CMS','p','p',16000.)
   CALL PYSTAT(2)

C...Book histograms.
   CALL HLIMIT(10000)
   CALL HBOOK1(1,'Higgs mass',50,275.,325.,0.)

C...Generate events; look at first few.
   DO 200 IEV=1,NEV
   CALL PYEVNT
   IF(IEV.LE.3) CALL LULIST(1)
   DO 100 I=1,N
100   IF(K(I,2).EQ.25) HMASS=P(I,5)
   CALL HF1(1,HMASS,1.)
   200   CONTINUE

C...Print cross-sections
   CALL PYSTAT(1)
   CALL HISTDO

END

Here 102, 123 and 124 are the three Higgs main Higgs production graphs $gg \rightarrow H$, $ZZ \rightarrow H$, and $WW \rightarrow H$, and MSUB(ISUB)=1 is the command to switch on process ISUB. Full freedom to combine subprocesses 'à la carte' is ensured by MSEL=0; ready-made 'menus' can be ordered with other MSEL numbers. The PMAS commands set the masses of the top quark and the Higgs itself, and the CKIN variables the desired mass range of the Higgs – a Higgs with a 300 GeV nominal mass actually has a fairly broad Breit-Wigner type mass distribution. The MSTP switches that come next are there to modify the generation procedure, in this case to switch off initial and final state radiation, multiple interactions among beam jets, and fragmentation, to give only the 'parton skeleton' of the hard process. The PYINIT call initializes PYTHIA, by finding maxima of cross-sections, recalculating the Higgs decay properties (which depend on the Higgs mass), etc. The
decay properties can be listed with PYSTAT(2). Inside the event loop, PYEVNT is called to generate an event, and LULIST(1) to list it. The information used by LULIST(1) is the event record, stored in the commonblock LUJETS. Here one finds all particles produced, both final and intermediate ones, with information on particle species and event history (K array), particle momenta (P array) and production vertices (V array). In the loop over all particles produced, 1 through N, the Higgs particle is found by its code (K(I,2)=25, see below), and its mass is stored in P(I,5). Finally, PYSTAT(1) gives a summary of the number of events generated in the various allowed channels, and the inferred cross-sections.

In the run above, a typical event listing might look like the following.

**Event listing (summary)**

<table>
<thead>
<tr>
<th>I</th>
<th>particle/jet</th>
<th>KF</th>
<th>p_x</th>
<th>p_y</th>
<th>p_z</th>
<th>E</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!p+!</td>
<td>2212</td>
<td>0.000</td>
<td>0.000</td>
<td>8000.000</td>
<td>8000.000</td>
<td>0.938</td>
</tr>
<tr>
<td>2</td>
<td>!p+!</td>
<td>2212</td>
<td>0.000</td>
<td>0.000</td>
<td>8000-8000.000</td>
<td>8000.000</td>
<td>0.938</td>
</tr>
<tr>
<td>3</td>
<td>!g!</td>
<td>21</td>
<td>-0.505</td>
<td>-0.229</td>
<td>28.553</td>
<td>28.558</td>
<td>0.000</td>
</tr>
<tr>
<td>4</td>
<td>!g!</td>
<td>21</td>
<td>0.224</td>
<td>0.041</td>
<td>-788.073</td>
<td>788.073</td>
<td>0.000</td>
</tr>
<tr>
<td>5</td>
<td>!g!</td>
<td>21</td>
<td>-0.505</td>
<td>-0.229</td>
<td>28.553</td>
<td>28.558</td>
<td>0.000</td>
</tr>
<tr>
<td>6</td>
<td>!g!</td>
<td>21</td>
<td>0.224</td>
<td>0.041</td>
<td>-788.073</td>
<td>788.073</td>
<td>0.000</td>
</tr>
<tr>
<td>7</td>
<td>!HO!</td>
<td>25</td>
<td>-0.281</td>
<td>-0.188</td>
<td>-759.520</td>
<td>816.631</td>
<td>300.027</td>
</tr>
<tr>
<td>8</td>
<td>!W+!</td>
<td>24</td>
<td>120.648</td>
<td>35.239</td>
<td>-397.843</td>
<td>424.829</td>
<td>80.023</td>
</tr>
<tr>
<td>9</td>
<td>!W-!</td>
<td>-24</td>
<td>-120.929</td>
<td>-35.426</td>
<td>-361.677</td>
<td>391.801</td>
<td>82.579</td>
</tr>
<tr>
<td>10</td>
<td>!e+!</td>
<td>-11</td>
<td>12.922</td>
<td>-4.760</td>
<td>-160.940</td>
<td>161.528</td>
<td>0.001</td>
</tr>
<tr>
<td>11</td>
<td>!nu_e!</td>
<td>12</td>
<td>107.726</td>
<td>39.999</td>
<td>-236.903</td>
<td>263.302</td>
<td>0.000</td>
</tr>
<tr>
<td>12</td>
<td>!s!</td>
<td>3</td>
<td>-62.423</td>
<td>7.195</td>
<td>-256.713</td>
<td>264.292</td>
<td>0.199</td>
</tr>
<tr>
<td>13</td>
<td>!c-!</td>
<td>-4</td>
<td>-58.506</td>
<td>-42.621</td>
<td>-104.963</td>
<td>127.509</td>
<td>1.350</td>
</tr>
<tr>
<td>14</td>
<td>(HO)</td>
<td>25</td>
<td>-0.281</td>
<td>-0.188</td>
<td>-759.520</td>
<td>816.631</td>
<td>300.027</td>
</tr>
<tr>
<td>15</td>
<td>(W+)</td>
<td>24</td>
<td>120.648</td>
<td>35.239</td>
<td>-397.843</td>
<td>424.829</td>
<td>80.023</td>
</tr>
<tr>
<td>16</td>
<td>(W-)</td>
<td>-24</td>
<td>-120.929</td>
<td>-35.426</td>
<td>-361.677</td>
<td>391.801</td>
<td>82.579</td>
</tr>
<tr>
<td>17</td>
<td>e+</td>
<td>-11</td>
<td>12.922</td>
<td>-4.760</td>
<td>-160.940</td>
<td>161.528</td>
<td>0.001</td>
</tr>
<tr>
<td>18</td>
<td>nu_e</td>
<td>12</td>
<td>107.726</td>
<td>39.999</td>
<td>-236.903</td>
<td>263.302</td>
<td>0.000</td>
</tr>
<tr>
<td>19</td>
<td>s</td>
<td>A</td>
<td>3</td>
<td>-62.423</td>
<td>7.195</td>
<td>-256.713</td>
<td>264.292</td>
</tr>
<tr>
<td>20</td>
<td>c-</td>
<td>V</td>
<td>-4</td>
<td>-58.506</td>
<td>-42.621</td>
<td>-104.963</td>
<td>127.509</td>
</tr>
<tr>
<td>21</td>
<td>ud_1</td>
<td>A</td>
<td>2103</td>
<td>-0.101</td>
<td>0.176</td>
<td>7971.328</td>
<td>7971.328</td>
</tr>
<tr>
<td>22</td>
<td>d</td>
<td>V</td>
<td>1</td>
<td>-0.316</td>
<td>0.001</td>
<td>-87.390</td>
<td>87.390</td>
</tr>
<tr>
<td>23</td>
<td>u</td>
<td>A</td>
<td>2</td>
<td>0.606</td>
<td>0.052</td>
<td>-0.751</td>
<td>0.967</td>
</tr>
<tr>
<td>24</td>
<td>uu_1</td>
<td>V</td>
<td>2203</td>
<td>0.092</td>
<td>-0.042</td>
<td>7123.668</td>
<td>7123.668</td>
</tr>
</tbody>
</table>

| sum: | 2.00 | 0.00 | 0.00 | 0.00 | 15999.98 | 15999.98 |

The event listing above is abnormally short, in part because some columns of information were removed to make it fit into this text, in part because all initial and final state QCD radiation, all nontrivial beam jet structure, and all fragmentation was inhibited in the generation. Therefore only the skeleton of the process is visible. In line 1 and 2 one recognizes the two incoming protons. In lines 3 and 4 are incoming partons before initial state radiation and in 5 and 6 after – since there is no such radiation they coincide here. Line 7 shows the Higgs produced by gg fusion, 8 and 9 its decay products and 10–13 the second step decay products. Up to this point lines give a summary of the event history, indicated by the exclamation marks that surround particle names (and also reflected in the K(I,1) code, not shown). From line 14 onwards comes the particles actually produced in the final states, first in lines 14–16 particles that subsequently decayed, which have their names surrounded in brackets, and finally the particles and jets left in the end. Here this also includes a number of unfragmented jets, since fragmentation was inhibited. Ordinarily, the listing would have gone on for a few hundred more lines, with
the particles produced in the fragmentation and their decay products. The final line gives total charge and momentum, as a convenient check that nothing unexpected happened. The first column of the listing is just a counter, the second gives the particle name and information on status and string drawing (the A and B), the third the particle flavour code (which is used to give the name), and the subsequent columns give the momentum components.

One of the main problems is to select kinematics efficiently. Imagine, e.g. that one is interested in the production of a single Z with a transverse momentum in excess of 50 GeV. If one tries to generate the inclusive sample of Z events, by the basic production graphs $q\bar{q} \to Z$, then most events will have low transverse momenta and will have to be discarded. That there are any events at all is due to the initial state generation machinery, which can build up transverse momenta for the incoming $q$ and $\bar{q}$. However, the amount of initial state radiation can not be constrained beforehand. To increase the efficiency, one may therefore turn to the higher order processes $gg \to Zq$ and $g\bar{g} \to Zg$, where already the hard subprocess gives a transverse momentum to the $Z$. This transverse momentum can be constrained as one wishes, but again initial and final state radiation will smear the picture. If one were to set a $p_T$ cut at 50 GeV for the hard process generation, events would be missed where the $Z$ was given only 40 GeV in the hard process but got the rest from initial state radiation. Not only therefore would cross-sections come out wrong, but so might the typical event shapes. In the end, it is therefore necessary to find some reasonable compromise, by starting the generation at 30 GeV, say, if one knows that only rarely do events below that fluctuate up to 50 GeV. Of course, most events will therefore not contain a $Z$ above 50 GeV, and one will have to live with some inefficiency. It is not uncommon that only one event out of ten can be used, and occasionally it can be even worse.

If it is difficult to set kinematics, often it is easier to set the flavour content of a process. In a Higgs study, one might e.g. wish to study the decay $H^0 \to Z^0 Z^0$, with each $Z^0 \to e^+ e^-$ or $\mu^+ \mu^-$. It is therefore necessary to inhibit all other $H^0$ and $Z^0$ decay channels, and also to adjust cross-sections to take into account this change, all of which is fairly straightforward. However, if one instead wanted to consider the decay $Z^0 \to c\bar{c}$, with a $D$ meson producing a lepton, not only would there then be the problem of different leptonic branching ratios for different $D$-s (which means that fragmentation and decay treatments would no longer decouple), but also that of additional $c\bar{c}$ pair production in parton shower evolution, to an extent that is unknown beforehand. In practice, it is therefore impossible to force $D$ decay modes in a consistent manner.

4.2 Standardization

While each program is pretty much a world in itself, some standardization effort has been started. The first step was to agree on a standard code for particles and partons, so that everybody uses the same numbers, contrary to the previous chaotic situation. A detailed description of this standard can be found in [48]. Here we reproduce a few important examples, to give the flavour of the scheme.

| 1 d | 11 e^- | 21 g | 111 | $\pi^0$ |
| 2 u | 12 $\nu_e$ | 22 $\gamma$ | 211 | $\pi^+$ |
| 3 s | 13 $\mu^-$ | 23 $Z^0$ | 311 | $K^0$ |
| 4 c | 14 $\nu_\mu$ | 24 $W^+$ | 321 | $K^+$ |
| 5 b | 15 $\tau^-$ | 25 $H^0$ | 2112 | $n$ |
| 6 t | 16 $\nu_\tau$ | | 2212 | $p$ |

The numbers for mesons and baryons, codes above 100, are built up from the flavour content, using the quark codes, and from the spin of the particle. An antiparticle has minus the code of the particle.
A next step is to agree on a common format for the events generated, i.e. how particle momenta, particle species, and event history are to be stored. A set of common-blocks were proposed for this purpose in [49]. Although these have not been included directly into all generators, translation routines to the standard common-blocks are found for many of the programs. Not only does this simplify the analysis of events obtained with different generators, but to some extent it also allows the event generation chain to be shared between programs. For instance, if one program can be used to generate $B$ mesons in some specific process, but is known to handle $B$ decays poorly, the $B$ decays could be inhibited there, and the undecayed $B$:s handed on to some other program.

The latest step in this evolution is found in [50], where also a standardization of particle decay data is proposed, among other things.

4.3 Program Limitations

Already in the previous section, we have considered some of the uncertainties in our current understanding of physics at the LHC. Many more examples could certainly have been found.

Another class of uncertainties comes from the presence of bugs, i.e. programming errors, in event generators. Given the complexity of LHC simulation, almost all programs have bugs. Some of these simply are typographical errors, others are correct transcriptions of incorrect formulae in the literature (e.g., the $WZ \rightarrow WZ$ matrix elements in PYTHIA were incorrect for several years because the published formulae were not correct), others are programs that work at current energies but break down when run in single precision at LHC energies, and yet others are real mistakes by the programmer. Given the size of these generators, an error can lie dormant for a long time before being discovered. Even when discovered, errors need not be correctly corrected by the authors. Indeed, in the recent LHC workshop we saw three such examples: the $gg \rightarrow \gamma \gamma$ matrix elements in PYTHIA, the $H \rightarrow \gamma \gamma$ partial width in ISAJET, and the $q\bar{q},gg \rightarrow b\bar{b}$ matrix elements in HERWIG. In each of these cases, the first 'corrections' proposed by the authors did not solve the problem found by users, and repeated complaints were necessary to see some improvements in the situation. Errors that were more rapidly corrected are too numerous to be mentioned.

These examples do not imply a quality judgement on particular programs. Considering the size and complexity, there is no reason to say that event generators are any more error-prone than other comparable software. The message is rather that all critical studies should always be based on more than one event generator, and/or on analytical cross-checks of the generator results.

With the changing computer market, e.g. the emergence of RISC chips, one must keep in mind that programs may need to be modified for maximum efficiency, or to be run at all. SAVE statements did not used to be necessary, but are it today. In the long run, programs are also likely to leave Fortran 77 behind, and move on to Fortran 90, C++, or some other more powerful language. To give one example where Fortran 77 forces us to unnatural solutions, consider how the properties of the event record are split into integer- and real-type arrays. What one would like to have, and what more modern languages do offer, is the ability to define new derived data types (structures) with both integer and real components. Each particle would then be represented by a vector, with integer components for particle code and history, and real components for momenta and production vertices. The event record itself would become an array of such vectors.

5 Summary

In this paper we have given an introduction to and overview of the LHC event generators currently available. As behooves a report of this kind, emphasis has been put on the unknown aspects. In particular, we have stressed the need for several independent cross-checks of crucial results.
However, one can also take another point of view: considering the number of years left before actual turn-on, the quantity and quality of LHC/SSC event generators are probably far superior to those available for any other major new accelerator at a corresponding stage of planning. We today have standard methods for turning the crank on any basic process (also including new hypothetical particles), to include initial and final state radiation, beam jets, fragmentation, etc., and to arrive at fairly realistic representations of what LHC events might look like. If the details may be a bit uncertain, the general picture of events at the LHC is still fairly clear. As experience from the Tevatron, LEP and HERA finds its way into programs, the quality should improve further. Needless to say, much continued work by event generator authors is necessary, not just to improve on the expected, but also to prepare for the unexpected.

REFERENCES

[8] F. James, Computer Physics Commun. 60 (1990) 329
[18] I. Hinchliffe, personal communication
[50] F. Carminati et al., in 'Large Hadron Collider Workshop', eds. G. Jarlskog and D. Rein, CERN 90-10 (1990), Vol. III, p. 52
Basics of Real-time Operating Systems

*Douglas Kemp*

CERN, Geneva, Switzerland

**Abstract**

These notes were used in a series of three lectures given at the CERN School of Computing at Ystad, Sweden, in August 1991. They deal briefly with interrupt handling, scheduling, memory management, intertask communication, input/output, file system, communications, standardisation and practical aspects. OS-9 is used as an example.

1 Introduction

The concept underlying real-time systems is that there will be a consequence if a required action is not taken within a given time after an external stimulus. The consequence may be more or less important:

- If an aircraft does not respond to its controls quickly enough you may die in the crash.
- If a robot is not given a command to stop it may hit a wall.
- If your physics data acquisition system misses an event it may not matter but you would be unhappy/concerned/fired if it missed too many.

Obviously the first two cases have something extra about them — namely that time is absolutely critical. Such cases are termed hard real-time whereas physics data acquisition systems are soft real-time.

In all cases response to external stimuli will depend on fast recognition that something needs to be done (interrupt response) and the time taken before starting to do what is needed (rescheduling). The ability to rapidly reschedule its work as a result of external stimuli is what differentiates a real-time system from a non real-time system. Naturally the speed (compute power) of the processor is important: the faster the processor the more that can be done in a given time. But unless the processor is rapidly rescheduled to be doing the required action at the relevant moment this extra processor power will be of no use.

These lectures will concentrate on soft real-time as needed for physics data acquisition systems. However there are hard real-time applications in experiments such as gas flow, access control.

2 The Operating System

The most common model used by programmers to describe the work to be done by the system is that of a co-operating set of tasks interacting with each other and the outside world via an operating system. The operating system provides the tasks with a virtual machine interface which:

- Hides the intricacies of the actual computer hardware (processor, discs etc) on which the system is to be run by providing a user-callable interface which is independent of the hardware.
- Lets each task run as though it were the sole task in the system by allocating the system's resources such as processing time, memory, input/output access etc in a controlled manner.
- Provides intertask communication mechanisms.
Operating systems usually consist of a kernel which provides the user interface to the basic operations and various agents (such as the file system) which make use of the kernel interface to provide higher level functional interfaces to the user.

A real-time kernel is characterised by:
- Fast response to external stimuli (Interrupt Handling)
- Rapid rescheduling of work
- Simple memory allocation schemes
- Mechanisms for communicating between co-operating tasks (Binary and Counting Semaphores, Events, Signals, Messages, Pipes and Shared Memory)
- Means for tasks to control the execution and priorities of tasks (including themselves)
- Ability to use clock time

Where detailed explanation of the functioning of an operating system is useful, I will take OS-9 as my example.

3 Interrupt Handling

In order to have a fast response to interrupts (low interrupt latency) the system must not lock-out interrupts for long periods. Interrupts are generally locked-out while the kernel is updating data structures which may be used from interrupt handling routines as well as user requests (cf the use of semaphores to protect critical regions detailed below). So to obtain a low interrupt latency the operating system and especially device drivers and their associated interrupt handling routines have to be written with this in mind. Another factor in the interrupt latency is the time needed by the hardware to change to the interrupt state. Although this is basically a function of the hardware, it may influence the design of the kernel (eg use of registers).

It is often important that interrupts have priorities. This allows a faster response to a higher priority interrupt, in the case where several interrupts occur essentially simultaneously.

Interrupt handlers can only use a restricted set of kernel calls. In particular they must avoid calls that might cause them to be queued. On exit from an interrupt the kernel must be called in case it needs to reschedule tasks.

4 Rescheduling of work

Each task is a single execution thread with an area of code, an area of data and a stack. When a task is running it will also need a register set, input-output capabilities etc. The static and dynamic information needed to define a task is commonly known as its context and will be kept in a Task Control Block (TCB).

The kernel must order the execution of tasks on the processor. This is known as scheduling. Typically a task will not be able to complete all of its work before it reaches a break (e.g. to wait for input). At this moment the kernel allows the execution of another task by switching from the present running task to another task which is ready to run. When it switches it must save the context of the running task, set up the context of the next task to run and then allow that task to run. This is known as context-switching. Obviously the context-switching time should be short for a real-time kernel. This will need careful minimisation of the context. Which task the kernel switches to (among those that are ready to run) is most often determined by giving each task a priority and then selecting that with the highest priority. However if one always had to wait for the running task to reach a break the response would be undefined and possibly very bad if the running task was doing a lot of computation. So what one wants to happen is that when a task of higher priority than that presently running becomes ready-to-run, usually due to an external interrupt, the running task is preempted and the new higher priority task is scheduled. This preemptive scheduling of tasks based on their priorities is the essence of
a real-time kernel. It means that a context switch is forced and has implications both for the context itself (which may have to include more information) and the individual tasks which may be stopped after any random instruction.

Although all real-time kernels offer priority-based preemptive scheduling this is not always enough. It can be very difficult to assign priorities to tasks, especially when they all have essentially the same priority eg a server. So the pure priority scheduling is often complemented by a scheme which allocates small units of time to all tasks with the same priority (time-slicing). A task is run for a slice (typically 20ms) and then put on the end of the queue for that priority, whilst the task at the head of the queue is scheduled.

OS-9 recognises that even this has its limitations if a high-level task uses all the processing time. The lower level tasks will be starved. To allow the lower level tasks to get a small fraction of the processing power one can age tasks. Every time one switches tasks the priorities of tasks in the ready queue will be increased. Eventually their priorities will reach a high enough level to ensure that they get a slice. After the slice they are reentered in the queue at their original priorities. This scheme is accompanied by a priority age limit so that really high priority tasks will not be affected.

5 Memory Handling

5.1 Memory Allocation

Most systems support kernel calls to obtain a contiguous block of memory of the required size from free memory pools maintained by the kernel. If there is insufficient contiguous memory available the task will wait until there is. This can result in tasks waiting for memory even though sufficient memory is available but it is not contiguous.

Total pool 10 units

T1 asks for 3 units (units 1-3 are given)
T2 asks for 4 units (units 4-7 are given)
T1 finishes and releases units 1-3
T3 asks for 4 units
a total of 6 units are available
but not 4 contiguous units so T3 is queued

This is known as fragmentation and also applies to discs if you are using contiguous sector allocation algorithms.

To avoid this problem it is best to allocate single fixed-size buffers from a memory partition. There can be several independent partitions with different fixed-size buffers.

5.2 Memory Mapping

When various compiled units are linked together to form a task, the linker will assume that the task is to be loaded in physical memory at a default address (eg 0). If several tasks are linked independently they would have to be given explicit, distinct addresses to avoid them overlapping when they were loaded into physical memory. Handling these addresses is obviously an error-prone activity when many tasks are considered.

There are software techniques for overcoming the need to give explicit load addresses (see later discussion of OS-9 module format). However it can be convenient to use a Memory Management Unit (MMU) to handle the problem. A loaded task may consist of several different segments of code and data. Each segment has an associated physical memory address and size. Address references generated by the running task, known as logical addresses are intercepted by the MMU and converted into physical memory addresses by
using a table containing the size and physical address of each segment used by the task (See Fig. 1).

![Diagram of Memory Mapping]

**Figure 1: Memory Mapping**

### 5.3 Memory Protection

The MMU can also check whether a logical address corresponds to a valid physical address or is outside the address ranges of all the segments of a task. If such an invalid address occurs an error is generated. This error can be used by a debugger to pick-up invalid pointer references etc. The information about each segment used by the task can also contain protection information such as read, write and execution permissions. The MMU can then check if the task has the relevant permission; if not generate an error for use by the debugger. OS-9 uses memory protection but no memory mapping.

### 5.4 Paging

If a segment is regarded as a contiguous area of physical memory it is possible to get memory fragmentation problems (as described above). One way of overcoming this is to build each segment from fixed-size pages of memory. The pages of a segment need not be contiguous pages of physical memory. Each segment will have a page-table giving the physical address of each page.

### 5.5 Virtual Memory

This is an extension of paging which caters for pages which are not in physical memory at all. Extra information is contained in the page-table saying whether the page is in memory or not and if not where to access it on the disc. A reference to a page not in memory will generate a page-fault and result in the task being suspended while the relevant page is loaded from disc. The program is then continued. This implies that the processor is capable of reexecuting an instruction which may have failed part-way through. When the physical memory is full of pages in use the system must have a mechanism for throwing out one page to enable it to be replaced by another. The generally used algorithm is Least Recently Used (LRU). The system will pick the LRU page which has not been written to (as a best choice). This is best as it avoids the need to write back the selected
page to the disc. To help it in this choice the MMU sets bits indicating that the page has been written-to or referenced.

Virtual memory can be used to simulate a memory bigger than the physical memory at the expense of reduced speed (since the memory is simulated on disc). It can also be used to load only the necessary parts of segments rather than the whole segment by allowing the page fault handler to do the loading rather than the loader itself. This is known as demand paging.

Although virtual memory would allow a program (or set of programs) of unlimited size to run in a given physical memory it is not used in real-time systems as the overheads involved and the uncertainties due to the paging are not acceptable. In addition a set of programs will generally need a certain amount of physical memory to run reasonably. If insufficient physical memory is available pages will be continually discarded from memory and new pages loaded from disc every few instructions. This process, known as thrashing, causes the performance to drop dramatically.

5.6 Module Structure in OS-9

OS-9 uses the concept of a module for code and data. The main requirements for code modules are that they are position-independent and do not modify themselves (reentrant). Position-independence allows OS-9 to load them from disc anywhere in memory and also implies that the compilers produce such code. The fact that they are reentrant allows the same piece of code to be shared between any number of tasks. Data modules may be loaded from disc or created dynamically in memory - the contents of data modules may, of course, be modified.

<table>
<thead>
<tr>
<th>Module Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Body</td>
</tr>
<tr>
<td>Cyclic Redundancy Check Value</td>
</tr>
</tbody>
</table>

Figure 2: Overall Module Structure

Each module consists of a header, a body and a cyclic redundancy check (See Fig. 2). The first part of a module header is common for all modules (See Fig. 3) whilst the rest of the header depends on the type of module (See Fig. 4).

The kernel keeps track of modules that are in memory using a module directory. Each such directory entry contains the name, address, size and the count of tasks that are using the module. This count is known as the link count.

When a task links to a module in memory, the module’s link count is incremented by one. When a task unlinks from a module the module’s link count is decremented by one. When a module is no longer needed (a link count of 0) its memory may be deallocated and it may be removed from the module directory.

When a module is to be loaded the following checks are made to ensure that the module is correct:
- Synchronisation Bytes OK?
- Header Parity OK?
- Cyclic Redundancy Check OK?
- Access Permissions OK?
- etc.
When the processor is booted from Read-Only-Memory (ROM) the bootstrap code checks the memory for modules by scanning for the Synchronisation Bytes and then checking it is has a real module. If so the module is entered in the module directory. Thus modules can be put anywhere in ROM. If two or more modules with the same name and type are found it will use that with the highest Revision Level. This allows updates to ROM without having to rewrite the existing module area.
6 Intertask Communication

6.1 Introduction

Not all kernels support all the mechanisms mentioned below. For any specific application one mechanism may seem more natural than another. Often one can build one mechanism using another but usually at the cost of efficiency as several calls to the kernel may be needed to do what a single call could have done.

6.2 Semaphores

When two or more tasks have access to a resource it may be necessary to ensure that only one may access the resource at a time. E. g.

Tasks T1 and T2 both wish to add 1 to a memory location x

code: load x ------a
    add #1 -------b
    store x ------c

say x has a value of 5
T1 is running
T1 reaches point (b) and is then interrupted
T2 runs -- it reads 5 and stores 6
T1 continues and stores 6 (NOT 7)

If the load, add, store had been a non-interruptible action (atomic), this problem would not have arisen. This piece of code is known as a critical region. Before entering a critical region one must obtain sole access and give-up the sole access right on exit from the region. Within a monoprocessor kernel, operations in critical regions can be made atomic by disallowing interrupts before entering the region and reallowing them on exit. This technique cannot be extended to user tasks which are generally not permitted to manipulate the interrupt mask. They should use a semaphore.

A semaphore has a value when it is free (say 1) and another (say 0) when it isn’t. There are two atomic operations, one to gain access (down) and another to release access (up):

down: if semaphore = 1 then semaphore = 0
    else add task to queue waiting on semaphore

up: semaphore = 1; if any task is waiting then allow
    the first task in the queue to run

Of course the down and up operations must themselves be made atomic by calling the kernel which can ensure this. These are Dijkstra semaphores. He used the names P and V instead of down and up but these are only meaningful to Dutch railway enthusiasts. The queue of tasks waiting on the semaphore could be arranged in First-In-First-Out (FIFO) order or task priority order (obviously better).

So to access a critical region one would use the following type of code:

semaphore critical-region-lock = 1;

P(critical-region-lock);
act-on-critical-region-data;
V(critical-region-lock);

The concept can be extended to give a counting semaphore by setting the initial value of the semaphore to n and decrementing/incrementing the semaphore value:
down : if semaphore > 0 then semaphore = semaphore-1
else add task to queue waiting on semaphore
up : semaphore = semaphore+1; if any task is waiting then allow
the first task in the queue to run

This could be used to handle a queue with a maximum of n entries with following
type of code:

semaphore queue-lock = 1, queue-count = 0, queue-space = n;

getitem()
{
    P(queue-count);
    P(queue-lock);
    get-item-from-queue;
    V(queue-lock);
    V(queue-space);
    return(item);
}

putitem()
{
    P(queue-space);
    P(queue-lock);
    put-item-in-queue;
    V(queue-lock);
    V(queue-count);
}

When one looks at the above code and sees that three semaphores are used and that
each function implies four calls to the kernel it is easy to see that the whole queue would
be better managed as a unit by the kernel. This would need only a single call to the kernel
for each of the above functions. So a message queue is born (see below).

The use of semaphores can very easily give rise to situations where two or more
tasks are waiting on each other (deadlocks) unless care is taken with the order in which
semaphores are "down'ed" and "up'ed":

T1: down (S1)
down (S2)

T2: down (S2)
down (S1)

T1 executes down (S1)
For some reason, gives up the processor to T2
T2 executes down (S2)
T2 executes down (S1) but gets put in the queue because S1 is owned by T1
T1 executes down (S2) but gets put in the queue because S2 is owned by T2

Another problem which can occur with semaphores (or any other shared resource) is
due to the priority of the task owning the resource being less than than that of the task
wishing to acquire the resource.
Example:

A low priority task (T1) obtains a semaphore.
A medium priority task (T2) starts to run whilst T1 waits in the ready queue.
A high priority task (T3) starts to run and tries to obtain the semaphore which T1 holds. But T1 can't run yet as T2 is ready and thus T3 must wait for T2 to complete and T1 to relinquish the semaphore.

To improve this situation T1 is allowed to run, until it relinquishes the semaphore, with T3's high priority. This is known as priority inheritance.

6.3 Events

Events are bits in (usually) a 32-bit group. There may be many such groups. Tasks pass a bit pattern to the kernel call. They can:
- Wait on any one of n events to occur (if not already set)
- Wait for all n events to occur (if not already set)
- Signal the occurrence of n events. This may satisfy the event wait condition for many tasks and hence functions as a broadcast.
- Clear n events

Examples: See Figs. 5 and 6

![Figure 5: Synchronising Tasks to an Interrupt](image)

![Figure 6: Synchronising Two Tasks](image)

6.4 Signals

Many tasks find it very useful to be able to handle asynchronous operations as they occur rather than having to execute a specific wait call to the kernel to find that something has happened. The asynchronous operations referred to may be expected (e.g.
data-ready on a communications line) or unexpected (e.g., error condition). Signals can be generated by the kernel, device drivers or other tasks. A signal can be regarded as a software interrupt. When a signal is received the task is interrupted and a signal handling subroutine, provided by the task, executed before the task is resumed. The signal handling subroutine can find out what caused it to be called, take appropriate action and possibly set some information in global variables for use in the task. Unlike events which may be received by any task signals are sent to a specific task or group of tasks.

If the receiving task has not indicated that it wishes to receive signals it will be aborted (except for WAKEUP [OS-9] — see below). The receiving task may disable/enable signal reception if it desires.

If the receiving task was not active it will be activated. The intercept code routine will be executed. The activation effect of a signal is not latched so if a signal arrives for an already active task, that task will not be reactivated when it next deactivates itself. E.g.:

\[
\text{if (anysignal==0) sleep(\text{forever});}
\]

would never awaken the task if a signal occurred at the point indicated. To avoid this one can disable signal reception before the test and use the fact that sleep and similar calls will automatically re-enable signal reception for you. In this case as soon as you issue the sleep call the task will be reawakened to handle the signal.

The kernel usually supports the following calls:
- Intercept: sets up the address of a code segment which will be called asynchronously when a signal occurs
- Send: sends a signal to a task or group of tasks
- Mask: enables/disables signals from reaching the calling task

A typical use of signals might be:

```c
int anysignal = 0;

sighand()
{
    anysignal = 1;
    various-actions;
}

/* set-up the signal handling routine */
intercept(sighand);

while ( )
{
    /* mask signals to avoid missing any signals occurring between the test and the sleep */
    mask(off);

    /* sleep will automatically unmask signals */
    if (anysignal == 0 ) sleep(forever)
        else mask(on);
    anysignal =0;
    process-signals;
}
```
OS-9 generates the following signals (which are queued if necessary — except for WAKEUP):

0 = KILL : unconditionally aborts the task
1 = WAKEUP : reactivates the task (if it was not already active).
              signal bypasses the intercept code and is not queued
2 = QUIT : CWL/E will send this signal to the last task to do I/O on the terminal
3 = INT : CWL/C will send this signal to the last task to do I/O on the terminal
4 = HUP : sent by the modem driver when the connection is lost (hang-up)

Signal codes 256–65536 are user allocated and may be generated by:
- Any user task
- The kernel ALARM calls (one-off or periodic)
- The I/O system when data is ready (option)
- The I/O system when the data carrier detect line changes (option)

6.5 Messages

Messages are blocks of information to be passed between tasks. Real-time kernels generally limit the message size to a fixed length to avoid buffer fragmentation. The contents of messages, except for a possible fixed header, are defined by the sending and receiving tasks and may include pointers to data areas etc.

Messages are sent to one of a number of exchanges for possible intermediate storage. Any number of tasks may send messages to a specific exchange. If a task is waiting for the message, that message will usually be copied directly to the tasks buffer bypassing the kernel buffers. It may also be possible to send a priority message to the front of the message queue or to send a message to all tasks waiting on the exchange.

A task wishing to receive a message will usually be suspended waiting on the exchange if no message is available. However the task may also be able to return immediately or after a given time with an indication that no messages were available. Any number of tasks may receive messages from a specific exchange. Waiting tasks may be queued in FIFO or priority order depending on how the exchange was created.

One could envisage:
- A data collection task putting a message into a queue for a data recording task.
- Later the system may be extended to have many data collection tasks putting messages into the queue for the data recording task.
- Then it may be found to be useful to have several data recording tasks (perhaps each using a different tape) extracting messages from the queue.

Messages can be used instead of semaphores, with the added attraction that they can also contain information. For example: if one had n printers. An initialisation task would create an exchange with n messages, the contents of which were the names of the printers. A task wishing to use any printer would get a message from the exchange, use the named printer, and then write the original message back to the exchange for further use by other tasks. Thus we have emulated a counting semaphore (with associated information).

6.6 Pipes

Pipes are an integral part of UNIX and have been included in some kernels. A pipe is a serial stream of characters connecting two tasks. It can also be used with multiple writers and/or multiple readers. The data in a pipe will usually consist of lines of text having the standard end-of-line terminator. Naturally the contents of each line are a matter for
agreement between the writer and reader tasks. OS-9 supports unnamed and named pipes. 
Unnamed pipes are used between a task (parent) and tasks it calls (children). Such pipes 
will disappear as soon as there is no path open to them. Named pipes will remain in 
existence until they are emptied by the last reader.

The typical use of unnamed pipes is to pass the output of one task into the input of 
a second task without the need for intermediate storage on a disc file. The two programs 
can then run in parallel, with the writer waiting when the pipe is full and the reader 
waiting when it is empty. e. g.

```
ls | wc -l
```

This will output the number of files in the present directory. 'ls' outputs the names 
of the files in the present directory, one per line. The character '|' is another hieroglyph 
meaning pipe the output of the task to the left into the input of the task to the right. 'wc 
-l' counts the number of lines in its input.

OS-9 makes extensive use of pipes since it is a UNIX look-alike. The Network File 
Manager (NFM) supports such pipes even across processor boundaries whilst retaining 
the full functionality including such options as generation of signals when data is ready. 
As NFM works with shared memory on back-planes, serial lines, Ethernet LAN's etc; such 
pipes are often used as a tool for building loosely-coupled distributed systems.

6.7 Shared Memory

This is a very good technique for high-speed data transfers of large blocks of data 
between tasks on the same processor or in a multiprocessor shared memory environment. 
The access to the memory will often be co-ordinated with messages (See Fig. 7).

![Diagram of Shared Memory]

---

Figure 7: Use of Shared Memory

7 Task Control (OS-9)

7.1 Creation and Deletion of Tasks

For OS-9 a task can be created by the fork or chain calls. When a parent task forks 
a child task the kernel will:

- Allocate a task control block (TCB)
- Search for the code module needed by the child task in the module directory. If the 
  module is not already loaded the kernel will locate the module on a file-structured 
  device (such as a disc), using the search path-list given by the parent, and load it 
  into a suitably sized chunk of memory. As OS-9 modules are position-independent, 
  the code module can be loaded anywhere in memory.
- Allocate a section of memory for the task's stack and data. Any initialised data is 
  set-up by taking a copy from the code module. In some cases, such as pointers to 
  code or data, the contents of the data area need to be relocated. The information 
  required to do this is contained in the code module.
• Set-up the parameter area with the string used when forking the task (e.g. mytask par1 par2) and make a copy of the parent’s environment data, I/O paths.
• Set-up the copies of the registers for the processor and Memory Management Unit which will be used when the task is run.
• Add the task into the ready-to-run queue.
  The parent and child will run in parallel. When the child terminates its data area and the modules it used will be released. The child’s TCB will not be removed until the parent has acknowledged the child’s termination. The chain command is equivalent to the parent executing a fork and then terminating itself - in this case the child’s TCB will be released immediately the child terminates.
  A task can also:
  • Suspend itself.
  • Resume a suspended task.
  • Change its, or any other task’s, priority.
  • Obtain the TCB details of itself, or any other task.

8 Use of Clock Time

  A task can:
  • Get the date and time with the resolution of a clock tick.
  • Set the date and time. Usually only done at system startup by a task which reads the information from a battery backed-up clock.
  • Inform the kernel that a clock tick has occurred. This is done by a special task associated with the clock interrupt.
  • Sleep for a specified time or forever. The task is put into a suspended state from which it will be awoken either when the time has lapsed or when a signal is received.
  • Ask for an alarm call. The task will receive a signal which it defines in a specified time or at a specified time of day. These alarms may be one-off or periodic, where the period is defined by the task. A typical application might be a task wishing to update a display every 10 seconds regardless of what else it had been doing.
  The kernel also provides time-out services for those system calls which implement them. A typical use would be in the recognition of probable deadlock situations.

9 Input/Output (OS-9)

  The I/O system is based on three types of module:

9.1 Manager

  When a task calls the kernel with an input/output request, the kernel routes the request to the manager concerned. Each manager handles all I/O for devices in a given class. The main managers provided by OS-9 are:
  • File Manager: any random access directory structured device. e.g. hard discs, floppy, RAM discs
  • Sequential Character Manager: e.g. terminals
  • Sequential Block Manager: e.g. tapes
  If a manager needs to do physical I/O it calls the relevant device driver.
  Each manager has the same basic structure namely a jump table with thirteen routines which may be called by the kernel:
  • Create
  • Open
  • Make Directory
• Change Directory
• Delete
• Seek
• Read
• Write
• Read Line
• Write Line
• Get Status
• Put Status
• Close

9.2 Device Driver

Each device driver can handle the physical I/O for all devices of the same type (e.g. using the same chip). It gets the device specific information that it needs, such as register addresses, from the device descriptor.

Each device driver has the same basic structure namely a jump table with seven routines which may be called by the manager:

• Initialise
• Read
• Write
• Get Status
• Put Status
• Terminate
• Error

It will also generally have an associated interrupt handling routine. When the driver’s initialise routine is called it will link this routine to the interrupt list for a particular interrupt vector (by calling the kernel).

9.3 Device Descriptor

The name a device is known by is the name of its device descriptor module. The device descriptor is a data module which contains the names of the manager and device driver modules for the device as well as information needed by those modules, such as the port address, interrupt level etc (See Fig. 8).

Each device must have a device descriptor. In some cases a device may have more than one descriptor (e.g. a floppy disc drive able to handle several different floppy formats).

10 File System (OS-9)

10.1 Structure

The file system is built up from files whose internal layouts/contents are defined by the tasks which will use them. One such task is the file manager itself. The files it uses, to give structure to the file system, are known as directory files. All other files are known as data files. The contents of directory files are the names of data files or other directory files and pointers to them. The overall result is a tree where the leaves are the data files (See Fig. 9). The root of the tree is a directory file known as the root directory and can be regarded as having the same name as the physical device. There is a root directory on each physical device so the file system is a set of trees; one tree per physical device. Directory file names are, by convention, written in upper-case letters whilst data file names are written in lower case letters.
11 Path Names

To refer to a data or directory file one follows a path from the root directory, via directories, to the required file. Using the character ‘/’ as a separator, the absolute path name for the file ‘golf’ would be:

/h0/DOUG/PLAY/golf

Figure 9: Example Tree

However one generally uses relative path names — that is path names relative to the present position in the directory structure. e.g. If the present position is ‘DOUG/PLAY’

```
golf
BRIDGE/deal
../..../myfile
```
[note: the construct ‘..’ takes you up one level in the tree and the construct ‘.’ refers to the present position in the tree]

When you login to a system it will position you at a directory in the tree before you start work. This position is known as your home directory and will often be used as a starting point when searching for files.

11.1 Implementation

File structured devices are regarded as an array of 256 byte sectors numbered sequentially from zero.

- Each file, be it data or directory, consists of:
  - A file descriptor sector
  - Any number of data sectors

A file descriptor sector (See Fig. 10) contains administrative information about the file and a segment table. Each segment table entry is a pointer, with an associated count, to a block of contiguous sectors on the device containing (part of) the file’s data. The segment table is used to chain the blocks of contiguous sectors into a logically contiguous file for the user task.

One of the bits in the administration area defines the file as being a directory file (See Fig. 11). This is an array of 32-byte entries. Bytes 0–27 are the name of the file and bytes 28–31 are a pointer to the file descriptor sector for that particular file.

Sector 0 of each device is an identification sector (See Fig. 12). It contains information about the device itself (total number of sectors etc), volume identification (name, creation date), a pointer to the bootstrap (if one exists) and a pointer to the file descriptor for the root directory.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>(D,S,PE,PW,RE,WE,W,R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner’s ID</td>
<td></td>
</tr>
<tr>
<td>Group and User</td>
<td></td>
</tr>
<tr>
<td>Date Last Modified</td>
<td></td>
</tr>
<tr>
<td>Link Count</td>
<td></td>
</tr>
<tr>
<td>Size (bytes)</td>
<td></td>
</tr>
<tr>
<td>Date Created</td>
<td></td>
</tr>
<tr>
<td>Segment Table [48]</td>
<td></td>
</tr>
<tr>
<td>Sector #, No of sectors</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: File Descriptor
Sectors 1–n contain an allocation map. This has one bit for each sector on the device. The bit is set if the sector is in use. This map is used by the file system manager when it wants to find free sectors or return used sectors to the free pool.

The bootstrap pointer is used when one starts up the processor. If it is zero the processor cannot be booted from this device.

The overall structure is illustrated in Fig. 13.

12 Communications

12.1 Introduction

Many control applications are conceived as groupings of heterogeneous computers. A typical present-day physics experiment has a number of computers collecting the data and passing it to a centralised data recording computer. This may well exchange information with computers in the accelerator control system, to obtain information on the state of
the beam, and a large number-cruncher for on-going analysis of a few of the events, to check on the quality of the data. Usually all the computers will be of different types. The number of computers in a typical experiment is growing rapidly and the need for flexible means of interconnecting heterogeneous computers is obvious.

Figure 13: Overall Structure

12.2 Local Area Network (LAN)

The solution adopted is to attach the computers to an Ethernet Local Area Network (LAN). Although there are limits to the number of connections and length of any segment of a LAN, these limits are of no concern to us since they can be overcome using various bridging techniques.

Ethernet is the most widely used LAN. It transmits, at 10Mbits/sec, frames of a defined format from one computer to either another computer, a group of computers (multicast) or all computers (broadcast). The contents of a frame are checked on reception but there is no check that a transmitted frame was received by the destination computer(s).
12.3 Software Protocols

To complement the basic hardware one needs a set of protocols to handle the exchange of data at the task to task level. The most common protocol suite in use is TCP/IP. TCP/IP is a spin-off from the DARPA research project with ARPANET. The Internet Protocol (IP) transfers a block of data (Internet datagram) between two computers on the same or interconnected networks. Every computer connected to the TCP/IP network anywhere in the world has a unique IP number which allows an access route to it to be generated. This is handled by IP. Above the basic IP service is the Transmission Control Protocol (TCP) which provides reliable inter-task exchange of byte streams using virtual circuits (cf a telephone conversation). Above IP there is also the User Datagram Protocol (UDP) which provides unchecked datagram transfers between any two tasks.

Above TCP and UDP are various other application oriented protocols. The principal ones are TELNET, File Transfer Protocol (FTP), the Socket Interface and Remote Procedure Call (RPC).

- TELNET allows one to login to a remote computer and use it as though one’s terminal/workstation was directly connected to it.
- FTP gives one the ability to read or write whole files on a remote file system.
- Sockets are like bidirectional pipes but function across heterogeneous computer networks. A task can create a socket and connect it to another socket anywhere else to exchange information with the task listening on the remote socket. Obviously the tasks will need a protocol for whatever they are going to do.
- RPC allows one to call a procedure as if it were local while in reality it is executed on another computer. This is done by calling a client stub, rather than the actual procedure itself. The client stub packs all the parameters up into a datagram which it sends to a server stub on the other computer. The server stub unpacks the parameters, calls the actual procedure and then returns the result in a datagram to the client stub. There it is unpacked and the result returned to the caller (See Fig. 14). As with all things in life there are complications such as different data formats on different computer architectures, making sure that the procedure is obeyed once only when network faults occur etc.

![Figure 14: Remote Procedure Call](image)

Above RPC is the Network File System (NFS). This provides transparent access to any part of files on a remote file system. In the usual UNIX implementation any part of a tree on a remote file system can be mounted at any leaf of a local file system (See Fig. 15).

12.4 Booting

Imagine the situation when you have a large number of discless (embedded) processors each having to execute a different program. You could have an EPROM set for each one but the work of updating the individual processors when a bug is found in some common software or even when you merely want to modify the code in a single processor is far from trivial. Also the management of the system as a whole can get complex. If the processors are connected by a LAN then they can use a small common boot package to load their code from a disc server and then run. On an Ethernet LAN each processor
has a unique Ethernet address, usually in a small EPROM. When the processor is booted it sends out a special broadcast packet containing its Ethernet address. This packet is recognised by a boot server which returns the processor's Internet number and the name of the file to boot from. The processor then loads the file and finishes the booting process.

Ethernet can also be used to force a reboot of a system. For this one needs extra hardware on the Ethernet card which will recognise a special packet code sent to that specific processor and generate a reboot sequence. This is very useful if the system is remote or inaccessible.

![Network File System Tree](image)

**Figure 15: Network File System Tree**

13 Standardisation

The costs of writing new software can be reduced by using as much existing software as possible and by using languages, communication protocols and operating systems which are familiar to the programming team. The same criteria will usually apply when transporting software to a new hardware environment as the software costs will far outweigh the costs of the new hardware. Thus the wish of most users and board manufacturers, is to have as much software standardisation as possible. Software suppliers, on the other hand, naturally tend to extol the virtues of their own solutions. However once the suppliers see that they will have to follow a standard if they wish to continue to exist they do so and usually benefit from the expanded market that the standard generates.

Obviously it would be best to have one kernel rather than the many incompatible systems available at the moment, assuming that implementations of it gave the required performance over a range of application areas. As most programmers are familiar with UNIX this is the clear candidate. If UNIX could be used in the real-time area then there would be many benefits such as savings in programmer training (they only need to know one environment), many existing packages could be used without having to rewrite them for each specific kernel and, most important, your application could be moved to a different hardware environment with minimal change. Unfortunately UNIX itself comes in many flavours. But there is a standards body working on this subject (POSIX 1003). However UNIX was not defined as a real-time system and so a committee in POSIX is defining the essentials of a real-time UNIX (1003.4). This work will need to be complemented by the provision of various profiles (e.g for embedded systems). A profile is a selection of the options and facilities offered by the full standard to fit a given need. For embedded systems one could envisage a system without virtual memory, minimal file system etc.
Another proposed standard which is being worked on and will be of interest in the real-time kernel world is OB IOS (P1256). OB IOS hopes to provide a low-level kernel-independent I/O interface to boards/chips. This should mask the real 'nitty-gritty', which is usually badly documented anyway, from the poor programmer, who can then handle the driver-level policy decisions rather than the gory details.

14 Practical Aspects

When one tries to pick a kernel for an application, one must bear in mind that the kernel is only part of a complete environment and you will need to consider other factors as well. For example:

- Does the system run on the CPU family(s) one might want to use? If not don't even consider it.
- Are relevant commercial boards available with a complete port (CPU, serial I/O, SCSI bus, networking ...)? If not what is the cost of port-packs for the various elements and how much will the port and subsequent maintenance cost in time/money?
- Do the other I/O boards have relevant drivers? If not how easy is it to write a driver — look at examples?
- If the system needs to be ROM'able is this a straightforward job?
- Is the required development environment available (hosted to your favourite workstation or native)?
- Are the compilers, linkers, debuggers, editors etc provided adequate? If not are there third-party or public domain solutions which fill the gaps at an acceptable cost in time/money?
- Will the code you write need to be maintained/updated over a long period of time? If so try and get a system with a strong commitment to standards.
- Is the price for the whole software environment acceptable? If not ...

15 Bibliography

1. "Operating Systems: Design and Implementation" Andrew S. Tanenbaum, Publisher : Prentice-Hall ISBN : 0-13-637331-3 025 (Not dedicated to real-time but has a good description of the various algorithms employed in operating systems )
2. Any manual for a real-time kernel that you can lay your hands on (e.g. OS-9, PSOS, VRTX, VXWORKS......)
OBJECT ORIENTED PROGRAMMING*)

Paul F. Kunz

Stanford Linear Accelerator Center
Stanford University, Stanford, California, USA

Abstract

This paper is an introduction to object oriented programming. The object oriented approach is very powerful and not inherently difficult, but most programmers find a relatively high threshold in learning it. Thus, this paper will attempt to convey the concepts with examples rather than explain the formal theory.

1. INTRODUCTION

In this paper, the object oriented programming techniques will be explored. We will try to understand what it really is and how it works. Analogies will be made with traditional programming in an attempt to separate the basic concepts from the details of learning a new programming language. Most experienced programmers find there is a relatively high threshold in learning the object oriented techniques. There are a lot of new words for which one needs to know the meaning in the context of a program; words like object, instance variable, method, inheritance, etc. As one reads this paper, these words will be defined, but the reader will probably not understand at that point the where and why of it all. Thus the paper is like a mystery story, where we will not know who’s done it until the end. My word of advice to the reader is to have patience and keep reading.

The first key idea is that of an object. An object is really nothing more than a piece of executable code with local data. To the FORTRAN programmer, an object could be considered a subroutine with local variable declarations. By local, it is meant data that is neither in COMMON blocks, nor passed as an argument. This data is private to the subroutine. In object oriented parlance it is encapsulated. Encapsulation of data is one of the key concepts of object oriented programming. The second key idea is that a program is a collection of interacting objects that communicate with each other via messaging. To the FORTRAN programmer, a message is like a CALL to a subroutine. In object oriented programming, the message tells an object what operation should be performed on its data. The word method is used for the name of the operation to be carried out. The last key idea is that of inheritance. This idea can not be explained until the other ideas are better understood, thus it will be treated later on in this paper.

An object is executable code with local data. This data is called instance variables. An object will perform operations on its instance variables. These operations are called methods. To clarify these concepts, consider the FORTRAN code in Fig. 1. This is a strange way to write FORTRAN, but it will serve to illustrate the key concepts. The style of capitalization is that which is recommended for objective programming, but for the moment is not important for the discussion. For this sample code, the name of the object is an object. The subroutine has two arguments. The first argument, msg, is used as the message, while the second, i, is used as an input or output parameter. This object has one instance variable with the name avalue which

*) Work supported by the Department of Energy, contract DE-AC03-76SF00515.
Subroutine anObject( msg, I )
Character msg(*)
Integer I
Integer*4 aValue
If ( msg .eq. "setValue" ) then
  aValue = I
  return
ElseIf ( msg .eq. "getValue" ) then
  I = aValue
  return
Else
  print( "0Error" )
EndIf
return
end

Fig. 1 Sample FORTRAN code

is of type integer. There are two methods defined, setValue, and getValue. What operations are performed on the data is defined in the FORTRAN statements. That is, if the value of the character string msg is "setValue" then the instance variable aValue is set to the value of the argument I; when the string is "getValue" then the current value of the instance variable is returned via I.

To send a message to anObject from some other FORTRAN routine, one might find code fragments that look like

Call anObject("setvalue", 2)
Call anObject("getValue", I)

In the first line, anObject will set its instance variable to value 2, while in the second line, the current value of the instance variable will be returned into the argument I.

Now the reader should have some of the key concepts understood, at least in their simplest sense. The variables that are local to a FORTRAN subroutine, that is, neither in COMMON nor passed as parameters, are encapsulated. The data is protected from being changed or accessed by another routine. It is this that makes it an object. An object has boundaries and clearly limits the accessibility of the variables in the routine. A FORTRAN COMMON on the other hand, has no boundaries. It is more like a fog. It spreads out from where you are and you don’t know where it ends. You also don’t know what else might be in the fog; or what you might run into.

Why we are programming this way is probably not yet apparent; that will come later. But for now, the reader should note the very different style of manipulating data. In languages like FORTRAN, we think of passing data to a routine, via arguments or COMMON blocks. Here the routine, i.e. the object, holds the data as instance variables and we change or retrieve the data via methods implemented for the object.

This messaging style of programming is a rather tedious way to get to the data that we want to operate on. Its time to invent a new syntax. An example of such a new syntax is the Objective-C[1] language, which is derived from the SmallTalk language. On most platforms, it is implemented as a translator that generates C code that works with a run time system to handle the messaging. Objective-C as a language is a proper super-set of C. It adds only one new data type, the object, and only one new operation, the message expression, to the base C language.

An example of Objective-C code is given in Fig. 2. This Objective-C code is equivalent to the FORTRAN code shown in Fig. 1. In the Objective-C syntax, the code is divided into two
#import <objc/Object.h>
@interface anObject:Object
{
    int aValue;
}
- setValue:(int) i;
- (int) getValue;
@end

@implementation anObject
- setValue:(int) i
{
    aValue = i;
    return self;
}
- (int) getValue
{
    return aValue;
}
@end

Fig. 2 Objective-C example

parts. The first part is called the interface; it is all the code between the @interface and the
next @end. The interface part of the code serves two purposes. It declares the number and type
of instance variables, in this case only one, and it declares to what methods the object will re-

dpond. The interface is usually placed in a separate file, then included via the standard C include
mechanism. Once again, the author can only say that the reasons for doing this are certainly not

apparent at this time, but will be explained later. The second part of the code is the implemen-
tation; this is all the code between the @implementation and the next @end. Within the imple-
mentation, one writes the code for all the methods that make up the object. Each method
begins with a "-" and the name of the method. Between the braces ("{" and """) can be any amount of
plain C code, including calls to C functions, and message expressions. Even calls to other com-

piled languages, such as FORTRAN can be placed here. The example in Fig. 2 admittedly
doesn’t show very much of that possibility.

To send a message to the above object, from another object, one might find the following

code fragments

    id anObject;
    int i;

    [ anObject setValue: 2 ];
    i = [ anObject getValue ];

In these fragments, anObject is declared to be data of type object, while i is declared to be
type integer. The message expression is signaled by an expression starting with the left bracket
("[") and ending with the right bracket ("]"). The syntax may seem strange to a FORTRAN pro-
grammer, or even a C programmer; it comes from the SmallTalk language. There is a lot more

behind it then can be understood now, so the reader would do best by not questioning it at this
point. For the remainder of this section, we’ll be using the Objective-C language for the exam-

ples so that we can study the object oriented concepts without needing to learn a completely new
language at the same time. Some readers may not be familiar with this language, or even the C
language. As an aid in following the text, the author offers Table 1., which crudely shows the
### Table 1. Correspondence between C, Objective-C, and FORTRAN.

<table>
<thead>
<tr>
<th>C</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;file&gt;</code></td>
<td><code>INCLUDE &quot;file&quot;</code></td>
</tr>
<tr>
<td><code>int i;</code></td>
<td><code>INTEGER I</code></td>
</tr>
<tr>
<td><code>float a;</code></td>
<td><code>REAL*4 A</code></td>
</tr>
<tr>
<td><code>int bins[100];</code></td>
<td><code>INTEGER BINS(100)</code></td>
</tr>
<tr>
<td><code>char title[80];</code></td>
<td><code>CHARACTER*1 TITLE(80)</code></td>
</tr>
<tr>
<td><code>char *title;</code></td>
<td><code>CHARACTER*80 TITLE</code></td>
</tr>
<tr>
<td><code>if ( x &lt; y ) {</code></td>
<td><code>a pointer</code></td>
</tr>
<tr>
<td><code>    }</code></td>
<td><code>IF ( X .LT. Y ) THEN</code></td>
</tr>
<tr>
<td><code>for ( i = 0; i &lt; n; i++ ) {</code></td>
<td><code>ENDIF</code></td>
</tr>
<tr>
<td><code>    }</code></td>
<td><code>DO I = 0, (N-1), 1</code></td>
</tr>
<tr>
<td><code>i++;</code></td>
<td><code>ENDDO</code></td>
</tr>
<tr>
<td><strong>Objective-C</strong></td>
<td><strong>FORTRAN</strong></td>
</tr>
<tr>
<td><code>#import &lt;file&gt; (include if not already included)</code></td>
<td><code>INCLUDE &quot;file&quot;</code></td>
</tr>
<tr>
<td><code>- set:(int) i</code></td>
<td><code>SUBROUTINE SET; ( I )</code></td>
</tr>
<tr>
<td><code>- (int) get</code></td>
<td><code>INTEGER I</code></td>
</tr>
<tr>
<td><code>- setLow:(float) x high:(float) y</code></td>
<td><code>INTEGER FUNCTION GET;</code></td>
</tr>
<tr>
<td></td>
<td><code>SUBROUTINE SETLOW:HIGH; ( X, Y )</code></td>
</tr>
<tr>
<td></td>
<td><code>REAL X, Y</code></td>
</tr>
</tbody>
</table>

correspondence between the Objective-C and C languages and FORTRAN. Of special note is the use of colons ("::") in the method names. In the first instance above, the colon seems to be a separator between the method name and the parameter; which it does except that the colon is also considered part of the method name. In the second instance in the table, there are two colons, each separating the method name from the parameters. The full name of the method contains the two colons as shown by the pseudo-FORTRAN code: `setLow:high:`. This is the style of the SmallTalk language and it is done that way for readability. It is very upsetting to FORTRAN programmers, but once one gets used to it, one begins to appreciate its self describing value.

So far, we’ve introduced a lot of new terms and a very different syntax. But what is important is the very different way of handling data. Where we are headed is probably not yet clear, but like I said in the beginning, this paper reads like a mystery story, we wouldn’t know until the end. I don’t want to lose you, so the next section will work on a much more concrete example using what we already are beginning to understand.

### 2. ANOTHER EXAMPLE: A HISTOGRAM OBJECT.

It’s time to take another example, something more concrete. I’ve chosen to treat a histogram as an object. We’ll examine the code to do one histogram. Figure 3 shows what the interface part might look like. The object shown is of the class `Hist` which inherits from the root class `Object`. The meaning of the words `class` and `inheritance` will be defined latter. The instance variables of the histogram object (shown between the braces) are the title, the low edge of the histogram, the bin width, the number of bins, etc. To make the example simple, we have a fixed maximum number of bins (100) and a fixed maximum title size. This is unnecessary in C, and thus Objective-C, because these arrays can be dynamically allocated when the histogram
#import <objc/Object.h>
@interface Hist:Object
{
  char title[80]; /* title of histogram */
  float xl, xw;  /* low edge and bin width */
  int nx;        /* number of bins */
  int bins[100], under, over; /* bins and under/overflows */
}
@end

- setTitle:(char *)atitle; /* set the title */
- setLow:(float) x width:(float) y; /* set the edge & width */
- setNbins:(int) n;         /* set the number of bins */
- acum:(float) x;          /* accumulate */
- zero;                    /* zero the histogram */
- show;                    /* print it */
@end

Fig. 3 Objective-C interface for Hist object

is defined, but for our present purposes, we'll avoid introducing features of the C language that are not available in FORTRAN.

Once the histogram object is created, the user would first send it messages to fix its title, set its low edge, bin width, etc. These messages might look like the following code fragments

[ hist setTitle:"my histogram" ];
[ hist setLow: 0 width: 1. ];

To accumulate and print, the messages might look like...

[ hist acum: x ];
[ hist show ];

The implementation of the histogram should be obvious. In the acum: method for example, one would find exactly the same kind of coding one would find in FORTRAN. That is, something like...

- acum:(float) x
{
  i = (x -xl)/xw;
  if ( i < 0 ) under = under+1;
  elseif ( i >= nx ) over = over + 1;
  else bins[i] = bins[i]+1;
  return self;
}

There is nothing but ordinary C code in this implementation. By the way, I've written the C code like a FORTRAN programmer might do, so as to not confuse the issue with short cuts a C programmer might normally use. If you're looking for something profound in all this, there isn't, yet.

It is rare that one wants only one histogram, so we now examine what needs to be changed to have more than one. First of all, if we have multiple histograms its clear that they all behave the same way. In object oriented parlance, we say there is a class of objects called histogram. In our example, the name of the class is Hist, as seen on the @interface line. The only difference between one histogram object and another is the values of its instance variables. Using the right object oriented words we would say that one histogram object is an instance of the class Hist. We create an instance of the class Hist by sending a special type of message to the class Hist. It is called the factory method. The messages that are sent to the class are factory meth-
ods. The ordinary messages are sent to an object, which is an instance of a class. Its is important to remember this distinction.

We send a message to the class to create an object, then we can start sending messages to the object. The code might look like...

```csharp
id aHist, bHist;
aHist = [ Hist new ];
[ aHist setTitle: "histo one" ];
bHist = [ Hist new ];
[ bHist setTitle: "histo two" ];
...
[ aHist acum: x ];
[ bHist acum: y ];
```

The first message, "new", is sent the class Hist. This is a factory method. All the classes that are linked together to form the program module are known at run time, just like the subroutines and functions are known in FORTRAN. Classes can only accept factory methods, so to distinguish them from objects, one capitalizes the first letter of the class name. Factory methods return the id of the object created. In the example, we've given these ids the names aHist and bHist. Once an object has been created, i.e. an instance of the class Hist, then we can send messages to the object to define the histogram, and accumulate into it. No other changes are needed to make multiple histograms. All the allocation of objects of class Hist, i.e. the memory management, is done automatically by the compiler. Part of this comes from the base C language, but the Objective-C factory methods make it even more transparent.

At this point the FORTRAN programmer is probably confused, since we have shown code which seems to be written for only one histogram, and yet we have many. What's going on? One way to understand it is to look behind the scenes and see how memory is being allocated, as shown in Fig. 4. We write code for the class Hist which contains the instance variables of the class, its normal methods, and maybe a factory method if it is not inherited. At run time, we message the class Hist with a factory method. This method allocates space in memory for the instance variables, and some pointers to the executable code. Thus each object of class Hist has its private copy of the instance variables, but pointers to the same code. The factory method re-

![Fig. 4 Allocation of memory for objects](image-url)
turns the `id` of the object just created. It is actually just a pointer to the object. We can then send messages to this object. The run time support uses the pointers to find the code. Program execution jumps to one of the methods we see in class `Hist`. As the code executes, it sees only the instance variables of the object to which we sent the message. The net result for the programmer is profound. He writes the code for the `Hist` class as if there is only one histogram allowed. In the driver code, however, as many histograms as needed can be created via the factory method, and the system does all the bookkeeping.

At this point, we need to explain more about factory methods and what they do. We have already seen that they will allocate memory space. It is also a place where one may initialize an object to a set of default values. For example, the following code fragment could be the factory method of our histogram object

```plaintext
+ new
{
    self = [ super new ];
    xl = 0.0;
    xw = 1.0;
    nx = 0;
    strcpy( title, "none" );
    return self;
}
```

The `+` before the method name distinguishes a factory method from an ordinary one, otherwise the implementation proceeds as normal. The full explanation for the first statement will have to wait until the next section. At this point we’ll only say that the message to `super`, a reserved name, does the memory allocation and returns another reserved name, `self`. This name, `self`, is how an object refers to itself. For example, the following two statements are equivalent

```plaintext
xl = 0.0;
self->xl = 0.0;
```

C programmers will recognize that `self` is a pointer to the memory allocated for the instance variables. The remaining statements in the factory method are straight C code and need no further explanation.

A class can have more than one factory method. This might be used, for example, to simultaneously create an object of a class and set some of the initial values as is done with the following code fragment

```plaintext
+ newWithTitle:(char *)aString
{
    self = [super new];
    strcpy( title, aString );
    return self;
}
```

Now compare the object oriented style of writing a histogramming code with what one usually finds in a FORTRAN implementation. If we had written FORTRAN code to handle only one histogram, and decided that we needed multiple histograms, the changes to the code would be extensive. First of all, the local variables that held the definition and bin contents would all need to become arrays, dimensioned by some maximum number of histograms allowed. We would probably put these arrays in a COMMON block and write one routine for each operation we wanted to perform on the histogram, corresponding to the messages in the object oriented
approach. One of the arguments in these routines would be some kind of identifier of which histogram the operation was to be performed. The identifier frequently is not just the index into the arrays, but some character string, so we would need to write a lookup table to find the index from the identifier. To allow the flexibility of using the package for a large number of histograms with few bins, or a few histograms with many bins without re-compiling, one would like to get away from fixed arrays in COMMON blocks. In its place we find a program allocating space in some large COMMON block for the bins and the definitions. The net result in the FORTRAN implementation is that the person who writes the histogram package writes a lot of bookkeeping code, probably more bookkeeping code than definition or accumulation code. Instead of methods within a class being held together, we have independent routines, related only, perhaps, by some naming conventions. The data, instead of being encapsulated, is exposed since it is in a COMMON block. In short, everything is inside out when compared to the object oriented approach.

Of course as a user of the histogram package, one doesn’t really care about the difficulty of the implementation. Object oriented programming, at first, seems to offer no benefit. It’s worth mentioning two items along these lines. The first is that what is provided with such packages might be limited by the implementation language. OOP technology frees the programer of a lot of tedious work so he can concentrate on providing a better product. And secondly, the object oriented technology applies itself equally well to physics code since there is a lot of bookkeeping code in dealing with tracks, vertices, etc. We’ll see examples in the latter part of this paper.

3. INHERITANCE

Another important aspect of object oriented programming is inheritance which has been alluded to already. Let’s start with an example. Let us define an object called Hist2, which will be a two dimensional histogram. The interface file might look like the code shown in Fig. 5. It is just like the Hist object in the previous section. We’ll assume that the show method prints a table showing the accumulation in each bin.

Note that Hist2 has some of the same method names as those previously defined in the Hist class. Does this mean that one can not have both Hist and Hist2 classes in the same program? No, one can use the same method names in many classes. This is called polymorphism and it allows one to write code that is much easier to understand by re-using the name space for both data and function. With a language like FORTRAN one can only safely re-use the name space for variables local to one subroutine or function. Attempting to have variables of the

```
#import <objc/Object.h>
@interface Hist2:Object
{
    char title[80];
    float xl, xx, yl, yw;
    int nx, ny;
    int bins[100][100], ...;
}
-setTitle:(char *)atitle;
-setXlow:(float)x Xwidth:(float)y;
-setYlow:(float)x Ywidth:(float)y;
-acum:(float)x and:(float)y;
-show;
@end
```

Fig. 5 Interface for Hist2 class
same name in two different COMMON blocks all too often leads to clashes when both COMMON blocks are needed in the same routine. Also, subroutine and function names must be unique in one program. From this, one can see that the methods in OOP are not just subroutines or entry points.

Now suppose we want to define another form of 2D histogram which shows its contents in 3D form with the Z axis being the contents of the bin, i.e. a lego plot. We’ll call this class the Lego class. We can write its interface file as shown Fig. 6. There is only one instance variable and two methods in the class Lego. The instance variable plotangle is the angle at which the x-y axis should be shown when displaying. The two methods are to set that angle and to plot the histogram.

So what happened to all the methods to define and accumulate the lego plot? They are inherited. Notice the @interface line in the code above. It says that the class Lego is a subclass of Hist2. The use of the word subclass is a misnomer. One should not confuse subclass with subset. In object oriented programming it doesn’t mean something smaller, it means something bigger. When one class is a subclass of another, it inherits all of its superclass’ instance variables and all of its methods. Thus the Lego class has all the instance variables of the Hist2 class and one additional variable: plotangle. It also inherits all the methods of Hist2 and adds one new one: setangle:. What about the show method? A subclass can either take an inherited method exactly as it is in its superclass, or it may over-ride it. Since the fashion that the Lego class displays its accumulation is very different from that of Hist2, the class Lego needs to over-ride the definition of the show method with one of its own. The use of the Lego object is just like any other object. That is, we might see something like...

```java
Lego = { Lego new };  
[ Lego setTitle:"this plot" ];  
[ Lego setXlow: 0. Xwidth: 1. ];  
...  
[ Lego setAngle: 45. ];  
...  
[ Lego acum: x and: y ];  
[ Lego show ];
```

Again, it’s worthwhile to look behind the scenes and understand how memory is being laid out. Figure 7 shows how memory is allocated after one lego plot object is created. The object aLego consists of a concatenation of the instance variables of the Hist2 class and the Lego class. It is as if the block of memory for storing the Hist2 instance variables has been extended to accommodate those of Lego. The isa pointer points to the code defined in the Lego class. That class also has a pointer to the code of the Hist2 class. Thus, when aLego is sent the message “setAngle:” the code defined in the Lego class is found. When aLego is sent the message “setTitle:”, the method is not found in the code for the Lego class. Instead, the code found in Hist2 class is executed, because of inheritance. On the other hand, when aLego is sent the
message "show", the method in the Lego class is executed, because the show method in the Lego class over-rides the one in the Hist2 class.

One result of inheritance is much less code modification when we want to add functionality. Lego performs everything that Hist2 does and more. The author of the Lego class never needs to look at the code for Hist2; he only needs to know the methods he wants to over-ride and can add his own new methods at will. It also works in the opposite direction, if Hist2 changes, then Lego only needs to be re-compiled. The lego plot needed an extra instance variable, plotangle. This variable was added to the class without needing to change anything in the Hist2 class to accomodate it.

One programming note should be mentioned now. Note that the interface file for Lego (Fig. 6) includes the interface file for Hist2. This is necessary so that when Lego is compiled, the compiler can know what is inherited. This is one reason why the interface for a class is kept in a separate file; the so-called header file. Another reason is dealing with messages. If in some object one has a message like

```java
[ aLego setAngle: 45. ];
```

then the compiler needs to know the type of the parameter (e.g. int, float, etc.) and the type of the return value, if any. Thus, in the source code of the object that sends the above message, the interface file of Lego must be made visible to the compiler by including the Lego class header file. In practice, the header file is a very good place to put documentation in the form of comments. In the ideal world, the user of an object would understand what operations are performed by the object just from well chosen method names. In practice, however, some documentation is necessary and in many cases, the user might have to read the implementation code as well.

To further illustrate the use of inheritance, let us consider another example. Take a drawing program such as MacDraw or Canvas on the Apple Macintosh. With the mouse, the user can pick from a number of pre-defined shapes such as circles, rectangles, polygons, etc. One can move and resize these shapes in the drawing window. Object oriented programming is a natural way to implement such a program. One could define a generic shape class from which one could sub-class each of the particular shapes. In the shape class, one would find instance variables such as the x-y position on the screen and the size of the shape. Thus the methods to re-position the shape on the screen, or change its size would be implemented in the generic shape class. Any particular shape such as a rectangle, however, would have its own class derived from the generic
shape class and contain its own drawing method. A rectangle would fill its height and width. Methods to set the line width and color, to fill the area or not, etc. would be found in the generic shape class.

The class hierarchy that one should use in an application is sometimes difficult to design. The association of physical objects with programming objects is frequently a good rule to follow, but is not always the case. For example, one could argue that a square should be a subclass of a rectangle since it is a kind of specialized rectangle. However, making one's class hierarchy in this manner only confuses the program design. If one tried to define a square class as a subclass of the rectangle class, then how does one decide what the length of the square's side should be? That is, which inherited instance variable, height or width, should one use? It is better to leave a square shape as an instance of the rectangle class; one which happens to have equal height and width.

Another example of confusion results in trying to make the rectangle class as subclass of a polygon. If the polygon class responded to a message to increase the number of its sides, what should the rectangle subclass do if it received such a message? It can't add one side to its shape because then it would no longer be a rectangle. Over-riding the inherited add side method to do nothing doesn't seem natural either. The answer is the rectangle class should not be sub-classed from a polygon class. The rectangle class is simple, it fills its area with a rectangle. It's the polygon class which is more specialized and complex. It must not only fill its area, but have an additional instance variable to know the number of sides it has.

Although arriving at a good class hierarchy for an application may be difficult, especially for the beginner, it is generally not difficult to change the hierarchy. That is, once the problem becomes better understood, one can move methods and instance variables up or down the class hierarchy. For example, if one discovered a set of classes that all shared the same or nearly same properties, then one could add a super class for this set in which those properties could be stored and the methods for operating on them defined. This super class would become a abstract class, like the generic shape class of a drawing program, and the subclasses would become more specific, like rectangles, circles, etc. of a drawing program.

4. GRAPHICAL USER INTERFACES AND OOP

An example of the use of object oriented programming for the graphical user interface toolkit is shown in Fig. 8. This example is the class structure of NeXTstep, the GUI developed for NeXT computers. In this figure we see that a button is implemented by the Button class, which is a subclass of the Control class. Since controls are visible on the screen, they are a subclass of the View class, and since all views might respond to mouse or keyboard input, they are a subclass of the Responder class. For those methods implemented in the Responder class, all subclasses of View, e.g. Control, Box, Text, and ScrollView classes, will behave the same way, since they inherit these methods. Classes which are sub-classed from the View class all know how to draw themselves inside a window or panel. Classes which are sub-classed from the Control class have the additional property that when they receive an event from mouse or keyboard, they may send a message to another object, even one that does not present itself on the screen. For example, a slider that is clicked and dragged, will send a message to some object with the current value of the slider as a parameter.

The use of the NeXTstep class structure can also illustrate another aspect of object oriented programming that one frequently makes use of. That is, an object can be composed of many different objects from different parts of the class hierarchy. The panel shown in Fig. 9, for example, is one used to display particle properties stored in a data base object. It was developed for the Gismo project[2] and later used within the Reason project[3] as well. Note already that this panel object is used by another object to display its values. This implies that the data base
object has an instance variable that is the id of the panel object. The panel contains buttons, text fields, scroll-able views, etc., each of which are also objects which are subclasses of Control and View subclasses. Thus the panel object for the application is made up of objects from various classes. The panel is effectively a container object which controls its position on the screen. It contains a number of view objects which draw themselves within the panel. Those that are also control objects can send messages between themselves, the panel, or the database object that owns the panel, depending on user input. Likewise, the database object can send messages to the panel object or any of the view and/or control objects contained by the panel. Each object

![Fig. 8 Graphical User Interface class hierarchy](image)

![Fig. 9 Panel object with imbedded view objects](image)
manages its own little world as defined by the instance variables of its class and its inherited instance variables. The application works by the objects sending messages to each other.

The lesson to learn from this complex panel object is to make the distinction between the class hierarchy and other hierarchies that might appear with their use. The former defines for a subclass what methods and instance variables are inherited, while the latter defines how various objects are linked together in an application. For example, the panel object contains a view object which holds its contents. Within that content view there are a number of different kinds of sub-views which may also contain sub-views. But each one of these sub-views derives their class definition from the same point within the class hierarchy.

5. OBJECT ORIENTED FORTRAN

So far we have given all the examples in the Objective-C language. This language is C with one new data type, an object, and one new expression, the message, compared to the C language. The Objective-C language was originally implemented as a pre-processor to the C compiler. It generates C code which is then compiled and linked to the Objective-C run time library. It was designed so that the syntax could be added to other languages as well.

Such a pre-processor can also be written for FORTRAN. For the NeXT computer, the Abssoft company has done exactly that in order that programs written in FORTRAN can make use of the NeXTstep class library. An example of object oriented FORTRAN code is shown in Fig. 10. One can tell this is FORTRAN source code because of the use of symbols like .false. One can also recognize that it uses the same syntax as Objective-C with statements like the message expressions. After passing this source code through a pre-processor, it is compiled by the FORTRAN compiler and linked with the standard NeXT libraries. Thus one has an existence proof of an object oriented FORTRAN. However, the current implementation permits only one instance of an object. This limitation will be removed in future implementation of the compiler.

```fortran
INCLUDE "appkit.inc" ! Include Application Kit
INCLUDE "Timer.inc"
INCLUDE "Cube.inc" ! Include the interface

@implementation Cube : View

@+ newView:REAL*4 rect(4)
self = [self newFrame:&rect]
[self setClipping:NO] ! This speeds drawing
width = 2.0 ! Start with line width of 2.0
suspend = .false. ! Start with cube rotating
[Timer newTimer: @0.02D0
+ target: self
+ action: Selector("display\0")]
newView_ = self ! Return, by convention, self
@end

@- step ! Suspend rotation. do a single step
suspend = .false. ! Temporarily turn off
[self display]
suspend = .true. ! Display new rotation of cube
step = self ! Suspend cube
@end
```

Fig. 10 Example of object oriented FORTRAN
It is the author's belief that an object oriented FORTRAN has more value to HEP than FORTRAN '90. Not that FORTRAN '90 doesn't have some interesting features, as we will see in the next section, but that FORTRAN falls short of being an object oriented language.

6. INTRODUCTION TO C++

There are a number of reasons for giving a short introduction to C++. Although C++ is the most widely used of all object oriented languages, its syntax is initially a bit difficult to understand. In this section, we hope to understand enough of this syntax, by making analogies with Objective-C, to (hopefully) take away a bit of the fear of C++. Besides supporting OOP, C++ also supports abstract data types with operator overloading. These are very useful features that are sometimes confused with object oriented programming, so we need to understand them to see how they differ from true object oriented programming. C++ is a very powerful and flexible language but its syntax can be rather tedious as a result. Only certain aspects of the language can be covered here and the reader is referred to the ample supply of books on the language for further information (see Refs. [4] to [6]).

An example of a C++ class header file is shown in Fig. 11. At first it looks pretty obscure. The class defined in the example is that of a 3-vector as it might be used in HEP. The class def-

```cpp
class ThreeVec {
    float x;
    float y;
    float z;
    public:
        ThreeVec( float px = 0.0, float py = 0.0, float pz = 0.0) {
            x = px;  y = py;  z = pz;
        }
        ThreeVec( ThreeVec& );
        inline float px(){ return x;  }
        inline float py(){ return y;  }
        inline float pz(){ return z;  }
        inline void setx( float px ){ x = px;  }
        inline void sety( float py ){ y = py;  }
        inline void setz( float pz ){ z = pz;  }
        inline void setVec( float px, float py, float pz )
        {
            x = px;  y = py;  z = pz;
        }
        inline float p2(){ return x*x + y*y + z*z;  }
        inline float p(){ return sqrt(p2());  }
        inline float pt2(){ return x*x + y*y;  }
        inline float pt(){ return sqrt(pt2());  }
        float phi();
        float theta();
        ThreeVec operator+( ThreeVec& );
        ThreeVec operator-( ThreeVec& );
        float operator*( ThreeVec& );
        friend ThreeVec operator*( float, ThreeVec& );
        friend ThreeVec operator*( ThreeVec& , float );
        friend ThreeVec operator-( ThreeVec& );
        void rotatePhi( float );
        void rotateTheta( float );
        /* etc ... */
}
```

Fig. 11  C++ header file to 3-Vector Class
inition consists of two parts: the class head with the keyword class followed by the class name, and the class body which is enclosed in braces ("{}"). This corresponds directly with the Objective-C interface section. In the class body, we have the definition of instance variables which are called data members in C++. The class ThreeVec has three data members; x, y, and z. They are followed by methods which are called member functions in C++. Unlike Objective-C, the member functions are not visible to objects of different classes unless they are explicitly declared to be visible. The keyword public has this effect. Instead of factory methods, C++ has special member functions called constructors which have more or less the same effect. A constructor member function is distinguished from the others by having the same name as the class. Thus, the function ThreeVec() is the constructor of the ThreeVec class. With these pointers, the header file shown in the example should start making a bit more sense.

In C++ each class is considered a new abstract data type which adds to the data types, such as float and int, that are built into the language. Thus, the ThreeVec class can be used just like the built-in data types can be used. In particular, allocating an object is done in the same manner as allocating any data type. The following code fragment illustrates this and how messaging is done in C++.

```cpp
int main()
{
    ThreeVec a;
    a.setVec(1., 1., 1.);
    phi = a.phi();
}
```

The variable a is of abstract data type ThreeVec and its space is allocated at compile time in the same manner as data types float or int. The member functions are invoked with the syntax that looks like accessing a data member of a C struct such as a.setVec() or a.phi(). The authors of C++ were concerned about run-time efficiency. Thus, inline functions were included in the language to allow access to data members while preserving their encapsulation. The keyword inline declares this as is shown in Fig. 11 for the member function setVec. In addition, the implementation of inline functions can be placed directly in the body, as is shown for setVec. This, however, is recommended only if the implementation is short.

The implementation of the member function phi() in the example is not inline. Thus it is usually done in another file, sort of like the Objective-C implementation file. A possible implementation is shown in the example below.

```cpp
float ThreeVec::phi()
{
    if (x == 0.0)
        if (y >= 0.0)
            return 0.5*M_PI;
        else
            return (-0.5*M_PI);
    float arctan = atan2(y, x);
    return arctan;
}
```

The double colon ("::") separates the class name from the member function name, otherwise the function is declared as one would in the C language. Also note in the code fragment that one can declare the type of a variable anywhere before it is first used as was done with the variable arctan, i.e. they do not need to be at the head of the function. Otherwise, the implementation of a C++ member function is pretty much like a C function. The inline declaration of new vari-
ables is one example of many improvements that C++ brings to the C language. There are many others, this paper is not the time nor the place to discuss them.

In C++, functions distinguish themselves from others with the same name not only by which class they are a member of, but also by what is called signature. The signature is the number and type of arguments and the return type. Thus the function int max(int, int) is distinct from float max(float, float). This aspect of the language is known as function name overloading. There are two constructor member functions in the ThreeVec class. Which one is invoked is controlled by the signature used. In addition, constructors may be invoked with a variable number of arguments and default values can be applied to missing arguments. The following code fragment shows four ways that a ThreeVec constructor might be invoked

    ThreeVec a;  // sets a to 0, 0, 0.
    ThreeVec b(1.);  // sets b to 1, 0, 0.
    ThreeVec c(1., 1., 1.);  // sets c to unit vector
    ThreeVec d = c;  // invokes ThreeVec( ThreeVec& )

The last method is similar to allocating and initializing a built-in type (int i = 1;).

To complete the data abstraction aspect of the language, C++ also supports operator overloading. An operator is simply that character symbol in the language that causes an operation to be performed on one or more operands. For example, the plus character ("+") is an operator to invoke the addition of quantities. Operator overloading means that for the abstract data types defined by the programmer, one can supply the meaning of the operators, i.e. overload the operator with a new meaning. For example, the + operator for the ThreeVec class might be defined as follows

    ThreeVec ThreeVec::operator+( ThreeVec& b )
    {
        ThreeVec sum;
        sum.x = x + b.x;
        sum.y = y + b.y;
        sum.z = z + b.z;
        return sum;
    }

Reading the first line above can be confusing at first. Let's decipher it. The function name is "operator+". It is a member function of the class ThreeVec as seen by double colons ("::"). It takes one argument which is a reference to a instance of the ThreeVec class, or we could say that the argument is of type ThreeVec. The ampersand ("&") indicates the argument is passed by reference; without it the argument would be passed by value. It also returns a value of type ThreeVec which is indicated by the first "ThreeVec" on that line.

One might be immediately puzzled about what happened to the other operand for the operator "+". It turns out that the instance of the class, i.e. the object which was messaged, will be the other operand, and it will be the operand on the left of the + sign. With this knowledge, we can see how the implementation works. When we see the use of the data member x, for example, it means the data member of the object on the left of the + sign.

At this point it is worth noting that an object of a class has direct access to the instance variables of other objects of the same class. This would seem to violate the protection of the data. The main purpose of protecting the data is to hide the structure of the data, i.e. where and how it is stored, so that should one change the structure, other classes don't need any changes. But if we change the structure of the data in a class, that class is going to be re-compiled anyway, so there's no harm giving direct access to members of the same class.

There is another mechanism to implement operator overloading which must be used when the left operand is of a type that one does not have control over. For example, if one wants to
multiply our ThreeVec by a float, with the syntax that looks like

```cpp
ThreeVec v1, v2, v3;
float a, b;
...
v2 = a*v1;
v3 = v1*b;
```

then the type float may be the left operand. The friend mechanism was invented to handle this situation. It allows functions which are not members of a class to have access to the protected data members of the class. In our ThreeVec example, the overloaded operator * was implemented this way. It was declared as a friend non member function in the class body. The implementation might look like

```cpp
ThreeVec operator*( float& a, ThreeVec& b)
{
    ThreeVec mul;
    mul.x = a * b.x;
    mul.y = a * b.y;
    mul.z = a * b.z;
    return mul;
}
```

Note that only because this non-member function has been declared to be a friend of the ThreeVec class (cf. Fig. 11) can it have access to the protected data members of that class. Note also that the non member function requires one more argument then the member function. One can use either the member function or non-member function methods of implementing operator overloading, but not both for the same class. This is because the compiler would find an ambiguity, which it could not resolve.

In both examples of operator overloading, the function returned a result of type ThreeVec. This object is in fact only temporary. When the function is invoked with a code fragment like

```cpp
d = a + b*c; // add and scale ThreeVec's
```

the object returned is the temporary the compiler must produce to form the result. The operator "\=" for the class ThreeVec must also be overloaded by a member function of the form

```cpp
ThreeVec& ThreeVec::operator=(const ThreeVec& a)
{
    x = a.x;
    y = a.y;
    z = a.z;
    return *this;
}
```

which does the assignment of d to the result. The compiler has this member function pre-defined for all abstract data types, but the programmer can over ride it by supplying his own. By the way, the this variable in C++ corresponds to the self variable in Objective-C, namely, its a reserved name for the object at hand.

The above description of adding two vectors may make it appear that there is a large overhead in the procedure. In fact, it is not. We are just looking at the step by step process a compiler must do to handle the addition of any two data types, even the built-in types. That is, operate on
two quantities and store the result as a temporary, finally make the assignment to the variable on the left of the equal sign by copying it from the temporary space. As with any good compiler, a good C++ compiler will optimize these sequences of operations.

There are many member functions in our ThreeVec example. Many of them simply access the vector in different ways. For example, in HEP one frequently deals with the transverse momentum. Thus when the ThreeVec class is used for momentum, we have the member functions float pt() and float pt2() to access the transverse and square of the transverse components of the vector. It may be tedious to implement all the variations that we may want to use. But by doing so, we will make the code that uses the ThreeVec class much easier and much more self describing. Most of these additional functions are declared inline thus there will be no lost in run time efficiency. If we did not include member functions like float pt(), then we would wind up implementing it explicitly each time we needed it. One could consider inline member functions of this type as something like a macro, but it is much safer then a macro because the compiler handles it.

Everything we’ve said about C++ in this section so far, is the data abstraction aspect of the language. Other languages that support data abstraction with operator overloading such as ADA and FORTRAN '90 have their own syntax for accomplishing more or less the same features. C++ distinguishes itself from these languages in supporting object orientation as well as data abstraction. To see this, consider the FourMom class shown in Fig. 12. This class is declared as a subclass of the ThreeVec class, thus it uses one of the important aspects of object oriented programming: inheritance. The keyword public in the class head is used to make public to the FourMom class the private members of the ThreeVec class. The other options, protected and private are too detailed to be discussed in this introduction to C++.

The FourMom class has one additional instance variable, or data member; the energy. Thus, the FourMom class differs from a vector with four components in that it is a Lorenz vector.

class FourMom : public ThreeVec {
    float energy;
    public:
    FourMom(float px = 0.0, float py = 0.0, float pz = 0.0, float pe = 0.0)
        : (px, py, pz) (energy = pe) {
    friend FourMom operator+( FourMom&, FourMom& );
    friend FourMom operator-( FourMom&, FourMom& );
    friend float operator*( FourMom&, FourMom& );
    inline float e(){ return energy; }
    inline void sete( float pe ){ energy = pe; }
    inline float p2(){ return energy*energy - ThreeVec::p2(); }
    inline float mass(){ return sqrt( p2() ); }
    inline void setFourMom( float px, float py, float pz, float pe ){
        sete( pe );
        setx( px );
        sety( py );
        setz( pz );
    }
    virtual void boost( double, double, double );
    virtual void boostToCMOF( FourMom& );
    virtual void boostFromCMOF( FourMom& );
    virtual void print( int, char* form = "%8.4f" );
    // etc. ...
};

Fig. 12 FourMom class declaration
It consists of a ordinary 3-vector component, which it inherits from the \texttt{ThreeVec} class and the energy component. Note how the member function \texttt{p2()} differs from an ordinary vector of four dimensions...

\begin{verbatim}
inline float p2() { return energy*energy - ThreeVec::p2(); }
\end{verbatim}

The use of the function name \texttt{p2()} for both the \texttt{ThreeVec} and \texttt{FourMom} classes is making use of the polymorphism that is available to us. To force the call to the \texttt{ThreeVec} function \texttt{p2()}, we had to use the scope operator (double colon ":初" ). As with the \texttt{ThreeVec} class example, we've put into the \texttt{FourMom} class many auxiliary member functions to make the class easy to use. For example, if we want to boost a four momentum to the center of mass frame of another four momentum we have the member function \texttt{void boostToCMof()} to do it.

Many of the member functions need to over-ride the inherited member functions of the same name, including the operator functions. While, if we want to get the transverse momentum of a four momentum vector, we have the inherited \texttt{float pt()} member function of the \texttt{ThreeVec} class available to us. This latter features brings out the difference between using the pure data abstraction and the object oriented approach. With data abstraction, a four momentum abstract data type might have a three vector component of it and thus look roughly like the same as objects with inheritance. However, with objects one inherits functions such as the transverse momentum functions, while with data abstraction alone, one would have to explicitly write code in the four momentum abstract data type to extract that information. Thus we see that data abstraction is quite distinct from object orientation even though for light weight objects, such as vectors, matrices, etc., they can be easily confused.

7. PARTICLE PRODUCTION MONTE CARLO WITH OOP

In this section the use of object oriented techniques for particle production and decay simulation will be studied. OOP is well suited for such simulations for many reasons. The creation of particles in the real world corresponds to the creation of objects in the programming context. Particle properties correspond to data members of a class. When a particle undergoes a physics process, it's just like an object responding to a message. Encapsulation allows one to store the particle properties and program implementation of physics processes together.

We'll look at prototype code from the MC++ project that is under development at University of Lund and SLAC[6]. The key idea in the MC++ project is to formulate the event generation chain as generalized "particles" decaying into other generalized "particles". For example, at LEP one would start with an "$e^+-e^-$-collision" particle which decays into a "$Z^0$" particle. It in turn decays into perhaps a "$q\bar{q}\text{-dipole}$" particle which may decay into a "$q\bar{q}\text{-string}$" particle. Further down the chain one would eventually create the hadrons and leptons which will also decay until we reach stable particles at the end of the chain. Each such generalized particle will have a list of ways it can decay and each element of the list will have a branching ratio, a list of decay products, and a pointer to a class which implements the decay.

Let's first look at the class inheritance hierarchy in which the particle class is imbedded as shown in Fig. 13. The \texttt{Particle} class is derived from the \texttt{FourMom} class that was discussed in the previous section. Thus energy and momentum aspects of a particle will be fully taken care of by its superclasses. The instance variables or data members in the particle class are divided into two groups. The first group describes the generic features of the particle type, such as charge, mass, a pointer to its decay channels, etc. The second group describes a particular instance of a particle such as its lifetime, creation values, a pointer to a list of its decay products, etc. Further specialization from the generic \texttt{Particle} class will be achieved by sub-classing. Thus one will handle the special attributes of strings, clusters, collisions, etc.
Figure 14 shows a code fragment of what the class definition might look like. Some interesting design choices are worth calling out. For example, the data member `charge` is stored as an integer with units of 1/3 the electron charge and the spin is stored as an integer with units of 1/2. To the theorists developing this code, these choices are very natural. However, to avoid confusion to an experimentalist who may want to access the charge, the inline function `charge` returns the charge of the particle in normal units.

Of particular note are the decay methods in the `Particle` class. Their implementations are shown in Fig. 15 and they are the heart of the code. The first method, `void decay()`, I call the theorist's method. It decays the particle into its children, then has each of the children decay. Since the children are particles, the decay method is invoked recursively. Since stable particles just return when the message is received (as seen by the first line of the code), recursion

```c
class Particle:public FourMom
{
    float mass;
    int charge;   // charge is in units e/3
    int spin;     // in units 1/2
    boolean isStable;
    DecayList* decayTable;
    /* etc... */
    Particle* parent;
    ParticleList childList;
    float lifeTime;
    boolean hasDecayed;

public:
    Particle();    // constructor
    inline float charge() { return (charge/3.); } // charge in units e/3
    inline int charge() { return charge; }
    inline float spin() { return (spin/2.); }  // spin in units 1/2
    virtual void decay();
    virtual ParticleList* decay();
    /* etc... */
}
```

Fig. 14 Particle class head and body
void Particle::decay() {
    if (isStable || hasDecayed) return;
    DecayChannel channel
        = decayTable->selectChannel(random.flat);
    (channel->decayer)->decay(this, channel->products);
    hasDecayed = YES;
    Particle* child = childList.top();
    while (child = childList++ ) child->decay;
}

ParticleList* Particle::decay() {
    if (isStable || hasDecayed) return;
    DecayChannel channel
        = decayTable->selectChannel(random.flat);
    (channel->decayer)->decay(this, channel->products);
    hasDecayed = YES;
    return childList;
}

Fig. 15 Decay methods of Particle class

ends with stable particles. Theorists normally don’t care about detector simulation, so this decay method suits their needs.

In the second line of code, a decay channel is created by randomly selecting one from the available channels in the decay table. The line of code says it about as well as my words. The decay channel object that one receives back from the decay table has a pointer to a decayer object. The third line of code invokes the decay member function of the decayer with two arguments; the particle being decayed and the list of decay products. The former is needed in order to access the four momentum and perhaps other properties. Obviously the latter is needed as well for similar reasons and to know what particles to create. The remaining lines of code in the decay() function does the recursion.

The second decay method I call the experimentalist’s method. This method decays the particle and returns the list of decay products. Thus, if the experimentalist is tracking a particle through a detector and finds that it needs to decay, he can invoke this decay method to get the decay products and then track them further through the detector. This method is identical to the theorist’ method with the recursive decays removed.

A key component of the MC++ program is the decayer class and their hierarchy. A representative sample of them is shown in Fig. 16. The main aspect of the decayer, as shown in Fig. 17, is that its decay() method is presented with two arguments; the first gives it the particle that is to decay and the second a list of particles that will be produced as has already been described above. The Decayer class is just an abstract place holder which is meant to be sub-
class Decayer {
    char* genericName;
    char* name;
    int number;

public:
    Decayer( char* name, int number );
    virtual boolean isAllowed( Particle* parent,
    ParticleList* children );
    virtual int decay( Particle* parent,
    ParticleList* children);
    ~Decayer();
    // etc.
}

Fig. 17 Decayer class body

classed by a class that represent a real physics processes. The keyword virtual tells the compiler about this. That is, that the function is meant to be over-ridden by a subclass. For example, the TwoBody class would implement pure kinematics for most particle decays. A ThreeBody decayer might just use phase space. Rather then re-writing 3-body phase space in C++, a ThreeBody class might be implemented by presenting the relevant parameters to a FORTRAN function or subroutine.

This latter suggestion highlights the difference and similarities between the object oriented approach and a procedural one like FORTRAN. At some low level, the physics calculations are the same; there's no way around that. C++ through its abstract data types allows for operator overloading, thus the calculation can be written in a much easier to understand manner. That is, adding two vectors has the same programming notation as adding two integers; its just like one would add two vectors on the blackboard, with a shorthand notation. Some non-object oriented languages support data abstraction as well. Data abstraction also changes dramatically the way such abstract data types are created and stored. The use of inheritance makes C++ also an object oriented language which further simplifies the way the programming to be done.

One important aspect of the MC++ program will not be shown here. It is the mechanism by which the properties of the generalized particles are initialized. It is done via a ParticleFactory class of which only one instance exists in the program. It is a reference table, or could be considered an on-line Particle Data Book for the known particles, and a reference model for the less well known or generalized particles. The ParticleFactory class supports reading in a reference table from disk and interactive modification of the table, in order to study particular decay modes or models by the user of the program. It is also responsible for setting up the list of Decayer classes that make up part of the program. When all these classes are done, the main program to simulate one e^+e^- collision might be as simple as the code shown in Fig. 18. This program decays one e^+e^- collision but could easily be extended to do more. It is also a prototype for code that would be put into a detector simulation program.

We have just walked through the basics of a particle generation simulation program written in C++ using both its data abstraction and object oriented features. Of course, we only looked at a prototype program and we didn't show a lot of the details, but nevertheless it appears that a full production quality program would retain the simplicity and modularity we have seen. One should also notice the complete lack of dimensioned arrays, only minor use of conditional statements, do-loops are almost completely hidden by the use of lists. All these attributes of object oriented code greatly clean up the implementation, allowing the author to concentrate on implementation of the algorithms at hand.
main()
{
    ParticleFactory factory; // invokes constructor
    // which reads disk file
    Particle p;
    p = factory.getA("e+e-collision");
    p.decay(); // decay it.
    p.print(); // to see results
    // that's all folks!
}

Fig. 18. A simple main program

8. SUMMARY

This paper has presented an overview of object oriented programming. The basic concepts
have been explored. The meaning behind word like instance variables, methods, member func-
tions, overloading, etc. has been explained. We have seen that although the style of program-
ing is very different, it is not inherently difficult.

There are many benefits of object oriented programming. Generally, programs using these
techniques are much more readable and maintainable. Also the code is more easily re-usable and
is generally very modular. In short, the goals of software engineering are far more easily
achieved with the object oriented approach. Compared to traditional programming, object ori-
ented code has fewer array declarations, thus minimizing the possibility of inadvertently ex-
ceeding array boundaries. Through the creation of objects, the system does the kind of
bookkeeping that one would need to do by hand in the traditional programming approach. In-
heritance makes is easy to modify and extend existing objects, while preserving the encapsula-
tion of data. Overall, it is much easier to implement large sophisticated programs.

Object oriented programming is made much easier when one starts with a class library
well suited to one's needs. For example, the NeXTstep class library is well suited for program-
ing an application with a graphical user interface. High energy physics will need a class library
for its specialized applications. It is hoped that out of projects such as Gismo[2], MC++[7], and
CABS[8] such a class library will develop.

In an age where one frequently talks of a "software crisis", the object oriented program-
mimg approach seems to offer some real solutions. Programmers and scientists who use the ob-
ject oriented techniques can be much more productive.
REFERENCES


[7] Richard Blankenbecler and Leif Lönnblad, Particle Production and Decays in an Object Oriented Formulation, Private communication, to be published.

Next Generation Transputers and Beyond – 1

Transputers and Routers:
Components for Concurrent Machines

Roger Shepherd, David May and Peter Thompson
INMOS Limited, Bristol, UK

1 Transputers

For some time now, VLSI technology has enabled a complete computer to be constructed on a single silicon chip. The Inmos T800 transputer [1], introduced in 1987, integrates a central processor, a floating point unit, four kilobytes of static random access memory plus an interface for external memory, and a communications system onto a chip about 1 square centimeter in area.

As a microcomputer, a transputer is unusual in that it has the ability to communicate with other transputers via its communication links; this enables transputers to be connected together to construct multiprocessor systems to tackle specific problems. Transputers are also unusual in that they have the ability to execute many software processes at the same time, to create new processes rapidly, and to perform communication between processes within a transputer and between processes in different transputers. All of these capabilities are integrated into the hardware of the transputer, and are very efficient.

The use of transputers for parallel programming has been greatly simplified by the development of the occam programming language [2]. The occam language allows an application to be expressed as a collection of concurrent processes which communicate via channels. Each channel is a point-to-point connection between two processes; one process always inputs from the channel and the other always outputs to it. Communication is synchronised; the first process ready to communicate waits until the second is also ready, then the data is copied from the outputting processes to the inputting process and both processes continue.

Each transputer has a process scheduler which allows it to share its time between a number of processes. Communication between processes on the same transputer is per-
formed using the local memory; communication between processes on different transputers is performed using a link between the two transputers. Consequently, an occam program can be executed either by a single transputer or by a collection of transputers connected in a network. Three different ways of using transputers to execute the component processes of a typical occam program are shown below.

This picture shows the same collection of occam processes executed on three different specialised networks. In the first network, which is a single transputer, each communication channel connecting two occam processes is implemented by the local memory of the transputer. In the other examples each channel is implemented by a physical link between different transputers.

Transputers have also been used to construct a number of general purpose machines, which all consist of an array of transputers connected together in a network. In some machines the network can be configured by software, for example by connecting the links via a crossbar switch. Many applications have been successfully ported to these machines and have demonstrated efficient parallel processing.

One of the problems with existing general purpose machines is that the need to carefully match algorithms to the interconnection networks of specific machines results in a lack of software portability. It has become clear that a standard architecture is needed for these general purpose message-passing machines. An attractive candidate is a collection of transputers connected by a high throughput, low delay communication network supporting communication channels between processes anywhere in the network.

2 Routers

There are many parallel algorithms in which the number of communication channels between processes on different transputers is much greater than the number of physical links available to connect the transputers. In some of these algorithms, a process executed on one transputer must communicate with processes on a large number of other transputers. These requirements for system-wide communication between processes are met by two new VLSI devices:

- the T9000 transputer [3] which includes hardware to multiplex many channels along a single physical link
- the C104 [3] message-routing chip (router) which can be used to construct efficient communication networks

This new communications architecture allows communication channels to be established between any two processes, regardless of where they are physically located. This simplifies programming because processes can be allocated to transputers after the program is written in order to optimise performance. For general purpose message-passing machines, a further benefit is that processes can be allocated to transputers by a compiler, which effectively removes configuration details from the program, thereby enhancing portability.

The use of two separate chips, one to perform computing (the transputer) and one to perform communication (the router) has several practical advantages:
Transputers can be directly connected without routers in systems which do not require message routing, so avoiding the silicon cost and routing delays.

It allows routers to have many links (e.g., 32 in the case of the C104) which allow large networks to be constructed from a small number of routers, minimizing the delay through the network. For example, 48 such routers can connect 512 terminals with only 3 routing delays, as in figure 1.

It avoids the need for messages to flow through the transputer, reducing the total throughput of the chip interface. This reduces the pin count, power consumption and package costs of the transputer.

It supports scalable architectures in which communication throughput must be balanced with processing throughput. In such architectures, it is known that overall communication capacity must grow faster than the total number of processors - a large machine must have proportionately more routers.

Since the new architecture allows all the external channels of a transputer to pass through a single physical link, system-wide communication can be provided by connecting each transputer to a routing network via a single link. The provision of several links on transputers (e.g., four links in the case of the T9000) allows each transputer to be connected to several different networks. Examples of the use of this technique are:

- The use of two (or more) identical networks in parallel to increase throughput
- The use of a main network and an (independent) monitoring and debugging network
- The use of a main network and an independent network for input and output (or for access to discs)

Another technique for increasing the communications throughput is to construct the network using two (or more) links in parallel for each connection. An example of a 2-dimensional network of this kind is shown in figure 3.

3 Virtual Links

In the initial transputers, each point-to-point physical link between transputers provides two communication channels, one in each direction. In the T9000 transputer, each physical link provides an arbitrary number of point-to-point virtual links. Each virtual link provides two channels, one in each direction. Hardware within the transputer multiplexes virtual links onto the physical links. At any moment, each physical link has an associated list of virtual links waiting to use it.

For each virtual link, a small control block is maintained in memory to record the state of the virtual link. When a process executes an input or output instruction on a virtual link, the process is descheduled and its identity is stored in the control block. At the same time the control block is used to determine the physical link to be used for the communication, and is added to the associated list of waiting virtual links.

The message is transmitted as a sequence of packets, each of which is restricted in length to a maximum of 32 data bytes. There are several reasons for this which are explained below. Each packet of the message starts with a header, which is used to route the packet to an inputting process on a remote transputer. The header also identifies the control block of the virtual link used by the remote inputting process. Thus a virtual link is established by setting up a control block in each of two transputers; the header in each control block is set so as to cause packets to be routed via the other control block.

Each packet of a message is transferred directly from the outputting process to the physical link and is transferred directly from the physical link to the inputting process, provided that a process is waiting when the packet arrives. An acknowledgment packet is then dispatched back along the virtual link as soon as each packet starts to arrive, so that transmission of acknowledge packets can overlap transmission of message packets. At the outputting end of the virtual link, the process will be rescheduled after the last acknowledgment packet has been received.
When the first packet of a message starts to arrive on a virtual link, it is possible that no process is waiting to input the message. In this case, it is essential that the packet is stored temporarily so that communication via other virtual links sharing the same physical link is not delayed. A single packet buffer associated with each virtual link control block is sufficient for this purpose, since the outputter will not send any further packets until an acknowledgement packet is received.

The splitting of messages into packets, each of which is acknowledged before the next is sent, has several important consequences:

- It prevents any single virtual link from hogging a physical link
- It prevents a single virtual link from hogging a path through a network
- It provides flow-control of message communication and provides the end-to-end synchronisation needed for synchronised process communication
- It requires only a small buffer to be used to avoid blocking in the case that a message arrives before a process is ready to receive it

4 Message Routing

Where system-wide message routing is required, a collection of transputers can be interconnected by a routing network constructed from routers.

![Clos network constructed from routers](image)

In some cases, it is convenient to construct a network from routers and attach transputers to its terminal links. An example is the clos network shown in figure 1. An alternative is to construct a network such as a hypercube or an array from a number of nodes, each node consisting of a transputer and a router as shown in figure 2.

Each router has a number of communication links and operates as follows:

- It uses the header of each packet arriving on a link to determine the link to be used to output the packet
- It arbitrates between two (or more) packets which must both be output through the same link, and causes them to be output one after another
- It starts to output each packet as early as possible (immediately after the output link is determined, provided that the output link is not already in use for another packet)

The overall throughput of the router is determined by the number of links which can be operating concurrently. An important benefit of employing serial links for packet routing is that it is simple to implement a full crossbar switch in VLSI, even for a large number of links. Use of a full crossbar allows packets to be passing through all of the links at the same time.
The ability to start outputting a packet whilst it is still being input can significantly reduce delay, especially in networks which are lightly loaded. This technique is known as *wormhole routing*. In wormhole routing, the delay through the switch can be minimised by keeping headers short and by using fast, simple, hardware to determine the link to be used for output.

The use of simple routing hardware allows this hardware to be provided for every link in the router. This avoids the need to share it between many links which would increase delay in the event of several packets arriving at once. Equally, it is desirable to avoid the need for the large number of packet buffers commonly provided in some packet routing systems (in which each packet is input to a buffer before output starts). The use of small buffers together with simple routing hardware allows a single VLSI chip to provide efficient routing between a large number of links.

The simple communications architecture allows a wide variety of implementations:
- CMOS VLSI can be used to construct routers with a large number of links. The C104 provides routing between 32 links, with routing hardware provided for each link. This is all incorporated in an easily manufacturable chip.
- It is straightforward to combine transputers and small routers on a single chip
- It is possible to construct routers in ECL or Gallium Arsenide technology to support high speed implementations of the link

For some purposes, it may be useful to combine a router together with each transputer in a single chip (or a single package). One example is the construction of a two dimensional array of simple transputers for image processing (for this application, no off-chip memory is needed, and most communication is local). The architecture of the routing system makes such a combination possible, as in figure 3.

4.1 Avoiding Deadlock

The purpose of a communications network is to support efficient and reliable communication between processes. Consequently, an essential property of a communications
network is that it should not deadlock. However, deadlock can occur in most networks unless the routing algorithm is designed to prevent it. For example, consider the square of four nodes shown in figure 4. Suppose that every node attempts to send a packet to the opposite corner at the same time, and that the routing algorithm routes packets in a clockwise direction. Then each link will become 'busy' sending a packet to the adjacent corner and the network will deadlock.

It is important to understand that deadlock is a property of the network topology and the routing algorithm used; it can also arise with buffered packet routing. In the above example, a single packet buffer at each node is sufficient to remove the deadlock but, in general, the number of packet buffers needed to eliminate deadlock depends on the network topology, the routing algorithm and the applications program. This is clearly not a satisfactory basis for the architecture of a general purpose routing system.

All of the above problems can be avoided by choosing networks for which deadlock-free wormhole routing algorithms exist. In such networks, buffers are employed only to smooth the flow of data through the network and to reduce congestion; often a buffer of size much less than the length of a packet is sufficient for this purpose. Most important of all, the buffering needed is not dependent on the network size or the applications program. It is possible to construct a single universal router which can be used for networks of arbitrary size and for programs of arbitrary complexity. An essential property of such a router is that, like a transputer, it can communicate on all of its links concurrently.

It turns out that many regular networks constructed from such routers have deadlock free routing algorithms. Important examples are trees, hypercubes and grids.
4.1.1 Trees are deadlock free

A tree consists of a collection of nodes with a single external link from the root. Assume that two trees $T_1$ with root link $r_1$ and $T_2$ with root link $r_2$ are both deadlock free; they will always perform internal communication without deadlock, and will accept and transmit packets along the root link without deadlock.

A new tree is formed by connecting the root links $r_1$ and $r_2$ to a new root node $R$; a further link $r$ on this node is the root link of the newly constructed tree $T$.

Any packet arriving at $R$ along $r_1$ is routed either to $r_2$ or to $r$. If it is routed to $r_2$, it will be consumed by $T_2$, because $T_2$ is deadlock free. If it is routed to $r$, it will eventually be consumed by the environment. By symmetry, packets arriving along $r_1$ will also be consumed. A packet arriving along $r$ will be routed to either $T_1$ or $T_2$; in either case it will be consumed because both $T_1$ and $T_2$ are deadlock free.

It remains to show that a tree with only one node is deadlock free; this is true because the node can send and receive packets concurrently along its single (root) link.

4.1.2 Hypercubes are deadlock free

To avoid deadlock in a hypercube, each packet is successively routed through the dimensions, starting from the highest.

A simple inductive argument can be used to show that this routing algorithm is free of deadlocks. Suppose that the order-$N$ hypercube is deadlock free. Combine two such order-$N$ hypercubes $H_1$ and $H_2$ to form an order-$(N + 1)$ hypercube by linking
corresponding nodes of $H_1$ and $H_2$. Any packet originating at a node $n$ in $H_1$ and destined for a node in $H_2$ will first travel along the link joining $n$ to the corresponding node in $H_2$; from this node it will be delivered by routing within $H_2$ and this is deadlock free by assumption. Similarly, any packet originating at a node $n$ in $H_2$ and destined for a node in $H_1$ will first travel along the link joining $n$ to the corresponding node in $H_1$; from this node it will be delivered by routing within $H_1$ and this is deadlock free by assumption. An important property of the node is that it is able to send and receive along a link at the same time; this is needed to ensure that a packet can flow from node $h_1$ in $H_1$ to the corresponding node $h_2$ in $H_2$ at the same time as a packet flows into $h_1$ from $h_2$.

It remains to show that the order-0 hypercube is deadlock free (which it is, being just a single node).

The effect of the routing algorithm can easily be understood in terms of the example shown in figure 4 above. Instead of routing all packets in a clockwise direction, the deadlock-free algorithm routes two of the packets anti-clockwise. This is illustrated in Figure 6.

![Figure 6: Avoiding deadlock in a simple network](image)

The fact that the hypercube is symmetrical means that the order of sequencing through the dimensions does not matter; it is important only that every packet is sequenced in the same order.

4.1.3 Arrays are deadlock free

The technique of routing a packet by systematically sequencing through the dimensions can be applied to any processor array. In fact, any rectangular processor array - whatever its size and dimension - is deadlock free! To prove this it is first necessary to establish that a line of processing nodes (a one-dimensional array) is deadlock free; this is guaranteed if a packet generated at a node takes the shortest path to its destination node.

A simple inductive argument similar to that used for the hypercube can now be used to establish that this routing algorithm is deadlock free.

4.2 Addressing

Every packet must carry with it the address of its destination; this might be the address of a transputer, or the address of one of a number of virtual channels forming input channels to a transputer. As a packet arrives at a router, the destination address must be inspected before the outgoing link can be determined; the delay through the router is therefore proportional to the address length. Further, the address must itself be transmitted through the network and therefore consumes network bandwidth.

It is therefore important that this address be as short as possible, both to optimise network latency and network bandwidth. However, it is also important that the destination
link can be derived from the address quickly and with minimal hardware. An addressing system which meets both of these requirements is interval labelling.

4.3 Interval Labelling

An interval labelling scheme [7] assigns a distinct label to each transputer in a network. For simplicity, the labels for an $N$ transputer network can be numbers in the range $[0, 1, \ldots, N - 1]$. At each router in the network, each output link has an associated interval - a set of consecutive labels. The intervals associated with the links on a router are non-overlapping and every label will occur in exactly one interval.

As a packet arrives at a router, the address is examined to determine which interval contains a matching label; the packet is then forwarded along the associated output link.

The interval labelling scheme clearly requires minimal hardware; a pair of comparators for each of the outgoing links. It is also very fast; at most a single comparison delay after the address has been input before the output link can be determined.

There remains the question of how to assign labels to an arbitrary network. The following examples give labellings for networks constructed from nodes as shown in figure 2.

4.3.1 Trees can be labelled

The transputers in a tree with $N$ nodes are labelled as follows. Suppose there are $L$ nodes to the left of the root node. Then the transputers to the left of the root are numbered $0, \ldots, L - 1$; the transputer of the root node is labelled $L$; the transputers to the right are labelled $L + 1, \ldots, N$.

Any node $n$ in the tree is itself the root node of a subtree $S$ with nodes $s_1, \ldots, s_h$. The interval associated with the left link of $n$ is $[s_l, \ldots, n - 1]$; that associated with the right link is $[n + 1, \ldots, s_h]$; that associated with the root link is $[s_h + 1, \ldots, s_l - 1]$. The interval $[s_h + 1, \ldots, s_l - 1]$ consists of all of the labels in the tree apart from those in $S$; numerically it consists of the two intervals $[s_h + 1, \ldots, N]$ and $[0, \ldots, s_l - 1]$. An example is shown in figure 7.

![A Tree with Interval Labelling](image)

4.3.2 Hypercubes can be labelled

The labelling of the hypercube follows the construction given for the deadlock free routing algorithm. In combining the two order-$n$ hypercubes $H_1$ and $H_2$, the transputers in $H_1$ are labelled $0, \ldots, 2^n - 1$ and those in $H_2$ are labelled $2^n, \ldots, 2^{n+1} - 1$. The link from each node $h_1$ in $H_1$ to the corresponding node $h_2$ in $H_2$ is labelled with the interval
\[ 2^n, \ldots, 2^{n+1} - 1 \] at \( h_1 \), and with \([0, \ldots, 2^n - 1]\) at \( h_2 \). This inductively constructs a hypercube together with the deadlock-free routing algorithm described above.

### 4.3.3 Arrays can be labelled

The labelling for an array follows the construction of the deadlock free routing algorithm. An \( n \)-dimensional array is composed of \( m \) arrays of dimension \( n-1 \), with \( m \) corresponding nodes (one from each \( n-1 \) dimensional array) joined to form a line. If each of the \( n-1 \) dimensional arrays has \( p \) nodes, the nodes in the \( n-1 \) dimensional arrays are numbered \( 0, \ldots, p-1; p, \ldots, 2p-1; \ldots; (m-1)p, \ldots, mp - 1 \). On every line the link joining the \( i^{th} \) node to the \( (i+1)^{th} \) node is labelled \([ip, \ldots, mp - 1]\) and the link to the \( (i-1)^{th} \) node is labelled \([0, \ldots, (i-1)p - 1]\). This inductively labels an array to route packets according to the deadlock free algorithm described above. An example is shown in figure 8.

![Figure 8: An Array with Interval Labelling](image)

### 4.3.4 Labelling arbitrary networks

The above labellings for hypercubes and arrays provide optimal routings; each packet takes one of the shortest paths to its destination. It can easily be shown that any network can be labelled so as to provide deadlock free routing; it is only necessary to construct a spanning tree and label it as described above. This clearly produces a non-optimal routing which cannot exploit all of the links present in the network as a whole.

Optimal labellings are known for all of the networks shown below:
- trees
- hypercubes
- arrays
- multi-stage networks
- butterfly networks
- rings

In high performance embedded applications (or in reconfigurable computers) specialised networks are often used to minimise interconnect costs or to avoid the need for message routing. In these systems, a non-optimal labelling can be used to provide low-speed system-wide communications such as would be needed for system configuration and monitoring.

### 4.4 Header Deletion

The main disadvantages of the interval labelling system are that it does not permit arbitrary routes through a network, and it does not allow a message to be routed through
a series of networks. These problems can be overcome by a simple extension: header deletion. Each outgoing link can be set to delete the header on every packet which passes through it; the result is that the data immediately following becomes the new header as the packet enters the next node.

Header deletion can be used to minimise delays in the routing network. To do this, an initial header is used to route the packet to a destination transputer; this header is deleted as it leaves the final router and enters the transputer. A second header is then used to identify the virtual link within the destination transputer. As the number of transputers is normally much less than the number of virtual links, the initial header can be short, minimising the delay through each router.

Another important use of header deletion is in the construction of hierarchical networks. In the 2-dimensional array of figure 3, each transputer could be replaced with a local network of transputers as shown in figure 9. Headers are deleted as packets leave or enter a local network. A single header can be used to route a packet within a local network, whilst three headers are needed to route a packet via the 2-dimensional array.

Figure 9: Local network of transputers and a router

4.5 Physical Link Protocol

Along every connection between a pair of communicating devices, it is important to ensure that data cannot be lost. This requires the introduction of control signals passing in the opposite direction to the data packets. It is sufficient to reserve a single flow-control token which signifies that up to $F$ fits (units of data) may be transmitted. Choosing a large value for $F$ allows continuous transmission of data even in the presence of long signal delays in the link - but requires a large buffer in the receiving link. A good compromise is to choose $F$ to be a few data tokens; in the T9000 and C104 $F=8$.

If a packet is sent directly from sender to receiver it is not necessary for the length of the packet to be represented within the packet; it is only necessary to ensure that the sender and receiver agree on the length. However, there are significant advantages in using a protocol which encodes the packet length as this provides for more efficient transmission of variable length messages.
Where packets pass through intermediate routers, it must be possible for these routers to determine the length of the packet passing through; this is required so that the (temporarily connected) path through the router can be disconnected as the end of the packet is output. Unless every packet is the same length, this requires that the protocol provides an indication of the packet length - a termination marker or an initial length count. The introduction of an initial length count would increase the packet delay through the router and also require logic to count through the packet.

The best technique is therefore to reserve a control token as the end-of-packet marker. If a further token is reserved as an end-of-message marker, it is possible to transmit variable length messages.

A complete protocol for a physical link requires the representation of at least the following:

- 256 data values
- 1 flow-control
- 1 end-of-packet
- 1 end-of-message

In the C104/T9000 system there are further tokens. For example, to ensure that disconnection of links can be detected, the protocol requires that transmission is continuous. There is therefore, a null token which is transmitted when no other token can be transmitted.

The C104/T9000 implementation uses 4 wire links to connect together two devices. Each link comprises 2 pairs of wires, each pair of which conveys tokens in a single direction. The tokens are transmitted serially, with one signal wire conveying the level (0 or 1), and the other making a transition when the level does not change between bits. This signalling convention can be decode without the use of very high speed clock. The initial bit of a token is a parity bit, there then follows a bit which distinguishes between data tokens and control tokens. For a data token, the distinguishing bit is a 0, and is followed by 8 data bits; for a control token the distinguishing bit is a 1, followed by 2 control bits, which classify the control token. This is shown in the table below, where P indicates the parity bit and D indicates a data bit.

<table>
<thead>
<tr>
<th>Data token</th>
<th>P 0 D D D D D D D D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow Control Token</td>
<td>P 1 0 0</td>
</tr>
<tr>
<td>End Of Packet</td>
<td>P 1 0 1</td>
</tr>
<tr>
<td>End Of Message</td>
<td>P 1 1 1</td>
</tr>
<tr>
<td>Escape Token</td>
<td>P 1 1 1</td>
</tr>
<tr>
<td>Null token</td>
<td>P 1 1 1 P 1 0 0</td>
</tr>
</tbody>
</table>

### 4.6 Summary

The routing algorithms described so far provide efficient deadlock free communications and allow a wide range of networks to be constructed from a standard router. Packets are delivered at high speed and with low latency provided that there are no collisions between packets travelling through the same link.

Unfortunately, for general purpose concurrent computers, this is not enough. In any sparse communication network, some communication patterns cannot be realised without collisions. Such collisions within the network can reduce system performance drastically. For example, some parallel algorithms require that all messages from one phase of a computation are delivered before the next phase starts; the late arrival of a single message delays all of the processors. In the absence of any bound on message latency it is difficult - and in many cases impossible - to design efficient concurrent programs. The problem of constructing general purpose concurrent computers therefore depends on the answer to the following question:

Is it possible to design a universal routing system: a realisable network and a routing algorithm which can implement all communication patterns with bounded message
5 Universal Routing

A universal routing system allowing the construction of scalable general purpose parallel computers was discovered by Valiant [4] in 1980. This meets two important requirements:

- The throughput of the network scales with the number of nodes
- The delay through the network scales slowly with the number of nodes ($O(\log(p))$ for $p$ nodes)

Notice that the aim is to maximise capacity and minimise delay under heavy load conditions - a parallel communications network is a vital component of a parallel computer. This is not the same as, for example, minimising delay through an otherwise empty network.

A $p$-node hypercube has a delay of $O(\log(p))$ if there are no collisions between packets. This is an unreasonable assumption, however, as all of the transputers will be communicating via the network simultaneously. An important case of communication is that of performing a permutation in which every transputer simultaneously transmits a message and no two messages head for the same destination. Valiant's proof [5] demonstrates constructively that permutation routing is possible in time $O(\log(p))$ on a sparse $p$-node network even at high communication load.

To eliminate the network hot-spots which commonly arise when packets from many different sources collide at a link in a sparse network, two phase routing is employed. Every packet is first dispatched to a randomly chosen intermediate destination; from the intermediate destination it continues to its final destination. This is a distributed algorithm - it does not require any central co-ordination - so it is straightforward to implement and scales easily. Randomisation does not, in fact, guarantee a delivery time less than $O(\log(p))$ - but it gives it a sufficiently high probability to achieve the universality result. The processors will occasionally be held up for a late message, but not often enough to noticeably affect performance.

5.1 Randomising Headers

How is a random intermediate destination chosen? As a packet enters a randomising network, it must be supplied with a random header; this header will be used to route the packet to a router which will serve as the intermediate destination. Each input link of a router can be set to randomise packets as they arrive. Whenever a packet starts to arrive along such a link, the link first generates a random number and behaves as if this number were the packet header. The remainder of the packet follows the newly supplied random header through the network until the header reaches the intermediate (random) destination.

At this point, the first (randomising) phase of the routing is complete and the random header is removed to allow the header to progress to its final destination in the second (destination) phase. The removal of the random header is performed by a portal in each router which recognises the random header associated with the router. The portal deletes the random header with the result that the original header is at the front of the packet, as it was when the packet first entered the network. This header is now used to route the packet to its final destination.

Unfortunately, performing routing in two phases in the same network makes the paths of the packets more complicated. The result is that deadlock can now occur.

5.2 Avoiding Deadlock

A simple way to avoid deadlock is to ensure that the two phases of the packet transmission use completely separate links. The node numbers are partitioned into two halves: one half contains the numbers used for the randomising phase. The numbers in
the other half are used for the destination phase. Similarly the links are partitioned into two sets: one set is used in the randomising phase and the other set in the destination phase.

Effectively this scheme provides two separate networks, one for the randomising phase, one for the destination phase. The combination will be deadlock free if both of the networks are deadlock free. The simplest arrangement is to make the randomising network have the same structure as the destination network - and to make both employ one of the known deadlock free routing algorithms.

Universal routing can be applied to a wide variety of networks including hypercubes and arrays [6].

6 Conclusions

Concurrent machines can be constructed from two components: transputers and routers. Transputers can be connected via their links to form dedicated processing systems in which communication takes place only between directly connected transputers. They can also be connected via routers allowing system-wide communication.

The provision of system-wide inter-process communication simplifies the design and programming of concurrent machines. It allows processes to be allocated to transputers after a program is written in order to optimise performance or minimise cost. It ensures that programs will be portable between different machines, although their performance will vary depending on the capabilities of the specific communications network used.

The communications architecture allows a wide variety of implementations. VLSI routers can provide routing between a large number of links, minimising network delays. Very fast routers with fewer links can be constructed using high-speed technology. Transputers and routers can be combined on VLSI chips to provide network nodes.

REFERENCES

Next Generation Transputers and Beyond – 2

T9000: superscalar transputer

Roger Shepherd
INMOS Limited, Bristol, UK

1 Introduction

A transputer is a VLSI component, consisting of a processor, a memory and a communication system, designed to be used in the construction of parallel systems. INMOS introduced the first transputer, the T414, in 1985. Since then a range of compatible 32-bit and 16-bit transputers have been introduced and these have been used in a wide range of applications.

The design of a transputer is undertaken on the assumption that it is a component of a multiprocessor system[4]. Many of the design tradeoffs, therefore, differ from those made by the designers of conventional microprocessors. These include devoting a significant proportion of the silicon area to of a communication system¹) and choosing an instruction set architecture tuned to the execution of concurrent programs, rather than sequential programs.

The T9000 is a new transputer[1]. It represents an improvement on the existing generation of transputer products in both capability and performance. The T9000 extends the transputer architecture in a number of ways. The most important of these is that the T9000 transputer decouples the physical connectivity of a system from its logical connectivity. Between any two directly connected T9000 transputers there may be established an almost unlimited number of virtual channels. The T9000 link system also enables transputers to be connected via a network of C104 packet routers which allows virtual channels to be established from any transputer to any number of other transputers. Other extensions of the architecture include the enhancement of the process model to provide per-process error handling facilities and the ability to run programs under memory management.

The T9000 has about ten times the performance of a T805. This improvement derives from a variety of sources including the use of caching, improvements in semiconductor technology, and a highly pipelined, superscalar processor.

2 Overview

The T9000 comprises a superscalar 32-bit processor (CPU) with a 64-bit floating point unit (FPU), a communications processor (VCP) together with 4 communication links (L), a control unit and its associated links (CU), an external memory interface (EMI) and 16K bytes of on-chip memory (CACHE). The final component of the T9000 is a crossbar switch (X-BAR) which connects the other components together.

The block diagram below shows the structure of the T9000.

1) 25% of silicon area for the T414, 16% for the T9000
3 Interconnection and memory structure

The key to the T9000 transputer's performance is in the memory architecture and internal interconnection structure of the device. This is important because of the enormous memory bandwidth that the T9000's subsystems demand. The majority of the bandwidth is demanded by the processor and the communication system. The T9000 processor, running at 50 Mhz, executes at about 10 times the speed of a T805 processor running at 20 Mhz. This means that the T9000 processor makes 4 times as many memory accesses per cycle as the T805 processor. A similar situation exists with the communication system. The four links of the T805 provide a maximum bi-directional data bandwidth of 11.2 Mbyte/s, whereas the four links of the T9000 provide a maximum bi-directional data bandwidth of 70 Mbyte/s. This means that the T9000 communication system makes 2.5 times as many data accesses per cycle. In fact, even more bandwidth than this is required as the memory holds data structures which support the virtual channel system.

Fortunately, a large proportion of the memory accesses made within a transputer are very structured and thus caching can be used to reduce the demand on the external memory system. The T9000 uses two levels of caching. The first level exists within the processor and provides caching of data in the workspace of the currently executing process. This has a significant effect as typically over 1/3rd of all access made by the processor are to the workspace. The second level of caching sits between the major functional blocks and the external memory system. Ultimately, however, accesses must be made to external memory and the memory interface of the T9000 provides 64-bit wide access, giving a maximum data rate from external memory of 200 Mbyte/s.

Even though caching reduces the requirement for external memory bandwidth there remains the matter of providing the processor and VCP with sufficient bandwidth to the cache. The T9000 adopts the classic solution of multibanking the memory system. The internal memory is divided into four banks, each bank caching one quarter of the address space. When an address is presented to the memory system the access request is routed through the cross-bar to the appropriate bank of memory. There are 9 ports onto the cross-bar, 1 for the PMI, 4 for the processor, 3 for the VCP and 1 shared between the scheduler and control unit. Each bank of the cache can provide access to one word every cycle and thus the total on-chip memory bandwidth is 800 Mbyte/s.

4 Cache

The T9000 has 16k bytes of on-chip cache. The cache is write-back and allocates when writes miss the cache. One advantage of this structure is that all accesses made to external memory are of complete cache lines. This removes the need to provide mechanisms to access individual bytes of external memory, even though the machine is byte addressed. (In fact, this is only partly true as it is necessary to provide for access to uncached regions of memory).

The cache is organised as 4 banks, each containing 4k bytes of fully associative memory arranged as 256 lines of 4 words. The banks are interleaved on address bits 4 and 5. This may seem strange as the obvious choice would be to interleaved on bits 2 and
3. However, that choice is incompatible with having multi-word cache lines containing contiguous addresses. The choice of four word cache lines represents a tradeoff between minimizing the external bandwidth generated by cache misses\(^2\), maximising the hit-rate\(^3\), and choosing an implementation architecture which is consistent with the chip area available. The practical choices for the T9000 were 2 words-per-line or 4 words-per-line. With a 64-bit wide memory interface the extra external bandwith of a 4 words-per-line cache, as compared with a 2 words-per-line cache is acceptable. The lower overhead (tag store) of the 4 words-per-line cache is attractive; if is probable that if a 2 word-per-line cache had been used then only 8k bytes of cache could have been implemented and this would have nullified any performance benefits that might otherwise have derived from the smaller line size.

Each bank of the cache is fully associative. Every 4 word line has associated with it a CAM which contains the 26 undetermined bits of address (2 bits are determined by byte within word, 2 bits by word within line and 2 bits by bank within cache). There are a number of benefits from using a fully associative cache as compared with a direct mapped or set associative cache. The first is that any address may be mapped into any line of the cache and so the cache hit-rate does not collapse when operating on data structures which “beat” with the cache size. A second advantage of using a CAM is that it is more power-efficient than a set associative cache. When a read is made from an N-way set associative cache, N lines are accessed, and for each line one word and the tag are read; the word from the line with the matching tag is then chosen. When a read is made from a fully associative cache only the single matching word need be read; this represents approximately a factor of N saving in power consumption. A further advantage of using a CAM is that a write can be made to the cache in a single cycle, rather than requiring first an access to the tag store and then a write to the data store. A novel feature of the T9000 cache, which is made possible by the use of a CAM, is that the one line is always kept empty, to be allocated when the next miss occurs. In this way the latency caused by a miss can be reduced as it is not necessary to first write back a (dirty) line before the cache line can be filled. The holding of this spare line is possible because any line can be used to cache any address. In an N-way set associative cache a similar mechanism would require keeping \(1/N\)th of the cache empty.

Thus far the internal memory has been described as a cache, however, it is possible to configure the memory so that it operates as pure RAM or as a mixture of 50% RAM, 50% cache. This is a useful facility for a number of purposes. One is that the T9000 can be used as a stand-alone computer with no external memory attached, a second is that it is possible to lock key areas of code and data into memory, either with a view to increasing performance, or with a view to improving predicability of execution time.

5 Processor

The T9000 was designed to provide a very fast implementation of an existing instruction set. Although the T9000 extends this instruction set, both to provide enhancements to the process model and to improve performance, to a first approximation the instruction set of the T9000 is the same as the instruction set of the T805. In order to understand the implementation techniques used in the T9000 processor it is necessary to understand the rudiments of the instruction set. For this purpose we will restrict ourselves to an explanation of how the transputer executes sequential code.

5.1 Instruction set architecture

The transputer has a small set of registers:

---

\(^2\) a smaller line typically generates lower external bandwidth

\(^3\) both a larger line and a larger cache typically generate a higher hit rate
The workspace pointer (Wptr) points to the workspace of the currently executing process. This workspace, which is typically organised as a falling stack, contains the local variables and temporaries of the process. Note that these reside in memory, rather than in a large set of registers. This is one architectural choice which enables the transputer to have a very fast process swap time.

The Areg, Breg and Creg are organised as stack. Loading a value into the stack pushes the value into Areg, pushes the old value of Areg into Breg and pushes the old value of Breg into Creg. Storing a value from Areg, pops the old value of Breg into Areg, pops the old value of Creg into Breg and leaves Creg undefined. The stack is used for the evaluation of integer and address calculations. The FPAreg, FPBreg and FPCreg form another stack, used for the evaluation of floating point expressions.

The instruction pointer points at the next instruction to be executed. Instructions are one byte long and each instruction contains 2 four bit fields:

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>OPERAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

The function field encodes the function to be performed and the operand field its operand.

There are functions to perform the most common operations of any computer program, loading and storing local variables (load local (ldl) and store local (stl)), loading constants (load constant (ldc)) and jumping (jump (j)). For example, load local, loads a word from the workspace, as specified by its operand, onto the integer evaluation stack. The instruction encoding provides for only 16 operands. Whilst it might be conceivable that programs could be implemented using only 16 local variables, the restriction of being able to jump only a maximum of 15 bytes is clearly a problem. To overcome this two of the function codes (prefix (pfix) and negative prefix (nfix)) are used to provide operand extension. Each prefix instruction prepends 4 bits of operand to the following instruction. In this way operands of any size can be constructed. The negative prefix instruction allows negative operands, which would otherwise require 8 bytes to encode, to be efficiently encoded. In this way, the transputer effectively has variable sized instructions, and in the following text we will normally implicitly assume the prefixing mechanism. Thus we refer to a ldc 17 instruction (and count it as a single instruction), when, in fact, this is a two instruction sequence, pfix 1; ldc 1.

The instruction encoding also appears to only allow for sixteen instructions. However,
a further function code (operate \(opr\)) is used to indicate that an operation is to be performed on the evaluation stack, and its operand selects the operation to be performed. For example, \(opr\ 5\), is \(add\) which adds the Areg and Breg together, putting the result into the Areg and popping the Creg into the Breg. The \(opr\) mechanism gives us 16 operations which can be encoded into a single byte. However, the prefixing mechanism can be used in conjunction with \(opr\) and thus the number of operations is not limited. As with the prefixing mechanism, we will implicitly assume the existence of the \(opr\) mechanism and will refer to instructions as \(remainder\) (\(rem\)) rather than as \(opr\ 31\) (or even \(pfix\ 1; opr\ 15\)).

As has been mentioned above, expressions are evaluated on the stack. There is no checking in the machine for stack overflow, this is easily done in a code generator which can arrange for extra temporaries to be introduced as needed. One advantage of this style of instruction set is that instruction encoding space is not wasted in continually respecifying register identities. Consider

\[
a := b + c + d
\]

on a transputer the code sequence is

\[
ldl b; ldl c; add; ldl d; add; stl a
\]  
(6 bytes, 6 instructions)

whereas on a 3-address register machine (assuming the variables reside in registers) we have

\[
\begin{align*}
  r(\text{temp}) & := r(b) + r(c) \\
r(a) & := r(\text{temp}) + r(d)
\end{align*}
\]  
(8 bytes, 2 instructions).

showing that the transputer code is denser. The sequence also illustrates that more transputer instructions than 3-address instructions may be required to perform the same operation. Note, however, that if, for the register machine, we had assumed that the variables resided in memory we would have had

\[
\begin{align*}
  r(\text{temp1}) & := \text{mem}(b) \\
r(\text{temp2}) & := \text{mem}(c) \\
r(\text{temp3}) & := \text{mem}(d) \\
r(\text{temp1}) & := r(\text{temp1}) + r(\text{temp2}) \\
r(\text{temp1}) & := r(\text{temp1}) + r(\text{temp3}) \\
\text{mem}(a) & := r(\text{temp1})
\end{align*}
\]  
(24 bytes, 6 instructions)

which requires as many 3-address instructions as transputer instructions. It should also be noted that in this case the 3-address machine requires four times the code space.

5.2 Sample instructions

It is useful to give the specification of a sample of the instruction set. This is not intended to be a comprehensive guide, but rather to explain enough of the instruction set that the design of the processor and the following examples can be understood. For a fuller description of the instruction set the reader is referred to [2].

5.2.1 Primary instructions

These are instructions which take an operand, designated \(n\) in the definitions. This is complete list of the primary instructions:
5.2.2 Secondary instructions: integer and addressing

These instructions are encoded via the opr mechanism.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>( Areg := Breg + Areg; ) pop ( Creg ) into ( Breg )</td>
</tr>
<tr>
<td>subtract</td>
<td>( Areg := Breg - Areg; ) pop ( Creg ) into ( Breg )</td>
</tr>
<tr>
<td>byte subscript</td>
<td>( Areg := Areg + Breg; ) pop ( Creg ) into ( Breg )</td>
</tr>
<tr>
<td>sixteen subscript</td>
<td>( Areg := Areg + 2 \times Breg; ) pop ( Creg ) into ( Breg )</td>
</tr>
<tr>
<td>word subscript</td>
<td>( Areg := Areg + 4 \times Breg; ) pop ( Creg ) into ( Breg )</td>
</tr>
<tr>
<td>load byte</td>
<td>load the byte addressed by ( Areg ) into ( Areg )</td>
</tr>
<tr>
<td>load sixteen</td>
<td>load the sixteen addressed by ( Areg ) into ( Areg )</td>
</tr>
</tbody>
</table>

There is no special load word instruction in the machine, its effect can be obtained by a ldnl 0 instruction.

5.2.3 Secondary instructions: floating point

The transputer has a second stack which is used for the evaluation of floating point expressions. This operates in broadly the same way as the integer stack. There is one important difference, which is that the floating point stack can hold both single precision and double precision numbers. Each of the registers in the floating point stack is tagged with the length of the number held in the register. There are two sets of load instructions, one for single precision, one for double precision. The actual floating point operations (such as floating point add) determine the type of their result from the type of their two operands. There are also two store instructions, one for single precision one for double precision. Although the length of the store operation could be determined from the FPAreg it is convenient to have the information encoded in the instruction.
The following are some of the floating point instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp load non-local single</td>
<td>fpldnlsn</td>
<td>push the single precision value at address Areg onto the floating point stack and pop the integer stack</td>
</tr>
<tr>
<td>fp load non-local double</td>
<td>fpldnldb</td>
<td>push the double precision value at address Areg onto the floating point stack and pop the integer stack</td>
</tr>
<tr>
<td>fp store non-local single</td>
<td>fpstnlsn</td>
<td>store the single precision value from FPReg into the address mem[Areg] and pop both stacks</td>
</tr>
<tr>
<td>fp store non-local double</td>
<td>fpstnldb</td>
<td>store the double precision value from FPReg into the address mem[Areg] and pop both stacks</td>
</tr>
<tr>
<td>fp add</td>
<td>fpadd</td>
<td>FPReg ::= FPBreg + FPReg; pop FPCreg into FPBreg</td>
</tr>
<tr>
<td>fp subtract</td>
<td>fpsub</td>
<td>FPReg ::= FPBreg − FPReg; pop FPCreg into FPBreg</td>
</tr>
<tr>
<td>fp multiply</td>
<td>fpmul</td>
<td>FPReg ::= FPBreg ∗ FPReg; pop FPCreg into FPBreg</td>
</tr>
<tr>
<td>fp divide</td>
<td>fpdiv</td>
<td>FPReg ::= FPBreg/FPReg; pop FPCreg into FPBreg</td>
</tr>
<tr>
<td>fp load and multiply single</td>
<td>fpldnlmulsn</td>
<td>equivalent to fpldnlsn; fpmul</td>
</tr>
<tr>
<td>fp load indexed single</td>
<td>fpldnlsni</td>
<td>equivalent to usub; fpldnlsn</td>
</tr>
</tbody>
</table>

The last two instructions above are provided to improve code density.

6 A superscalar implementation of the transputer instruction set

6.1 Workspace cache

The most important part of a fast implementation of the transputer instruction set is provision very fast access to local variables. In the transputer these reside in memory, whereas in a register machine a compiler attempts to keep them in registers. This puts the transputer at a potential disadvantage in two ways. The first is that access time to memory is slower than to registers, the second is that the transputer may require more memory bandwidth than a register machine.

These problems are overcome in the T9000 processor by using a workspace cache which can cache the first 32 locations of the workspace. This cache can be accessed essentially as fast as registers in a register machine, and can be used to supply data for all load local instructions which access the first 32 words of the workspace. The cache is operated as a write-through cache, that is, whenever a write is made to the workspace (for example by a store local instruction) the write is made both to the workspace cache and to memory. In this way memory always contains a correct image of the workspace, so that if a read is made to an address in the workspace, other than via a load local instruction, the correct value is loaded from memory.

6.2 Grouping

Another issue in implementing a fast transputer is achieving a fast execution rate. The conventional approach to speeding the execution of instructions is to use pipelining. However, the semantic content of many transputer instructions is low. This makes pipelining difficult as there is not enough work to do in these transputer instructions to warrant execution on a pipeline. To overcome this the T9000 groups several dependant instructions together and then executes the resulting group on a pipeline. In this way the T9000 can execute up to 8 instructions per cycle on a pipeline.
The T9000 pipeline is designed to implement efficiently the most common sequences of instructions produced by a compiler. For example, returning to a previously given sequence,

\[ ldl \; b; \; ldl \; c; \; add; \; ldl \; d; \; add; \; stl \; a \]

the T9000 executes this as two groups

G1: \[ ldl \; b; \; ldl \; c; \; add \]

G2: \[ ldl \; d; \; add; \; stl \; a \]

and the pipeline executes this as if it were performing the two operations

\[ Areg := Mem[Wptr + 4*a] + Mem[Wptr + 4*c] \]

\[ Mem[Wptr + 4*a] := Areg + Mem[Wptr + 4*d] \]

Assuming that the load local instructions can be serviced from the workspace cache, this sequence will execute in 2 T9000 cycles.

6.3 Pipeline structure

The T9000 pipeline has seven readily identifiable stages:

- The fetch and decode/group stages of the pipeline prefetch and then group instructions. The groups are then executed on the remaining 5 stages of the pipeline.
- The local stage is capable of performing two load local or load constant instructions each cycle. As the workspace cache is triple ported, with two read and one write port, it is able to support two load local instructions executing at the same time. Note that a load local instruction which references one of the first 32 locations of the workspace can be executed without having to compute the address of the workspace location; the operand of the instruction can be used to directly address the workspace cache.
- The address stage can perform two three-operand address calculations per cycle. These addresses may be used either in the non-local stage to read from memory, or in the write stage to write to memory. Note that for store local (unlike for load local) the address of the workspace location referenced has to be computed since a store will be made to memory as well as to the workspace cache.
- The non-local stage allows up to two reads at a time to be made from memory. Provided that the data reside in different banks of the cache the two reads can be made in the same cycle. No attempt is made to service these reads from the workspace cache; there is no need as the memory is kept consistent with the cache.
- The ALU/FPU stage is where arithmetic and floating point operations are performed. This stage contains hardware to perform shifts in 1 cycle, multiplication in 2 to 5 cycles, and normalisation in 2 or 3 cycles. The FPU is capable of performing single precision addition and multiplication in 2 cycles, and double precision multiplication in 3 cycles[3].
- Finally, writes are made in the write stage of the pipeline. Typically, the address will have been computed earlier, in the address stage, and the value to be stored will have been computed in the ALU/FPU stage. All addresses written to are checked to see whether they lie within the workspace cache, and if they do the workspace cache is updated, ensuring that it remains consistent with the memory.
Example of pipeline usage

This section illustrates the operation of the T9000 pipeline. We will look at the way the T9000 would multiply two single precision complex numbers. The C source for the operation is given below:

```c
typedef struct {float re, im;} complex;
complex *x, *y, *z;

x->re = (y->re * z->re) - (y->im * z->im);
x->im = (y->re * z->im) + (y->im * z->re);
```

Here $x$, $y$ and $z$ are pointers to complex numbers, that is, they point to pairs of single precision floating point numbers. The C code given calculates the product, $x$, of two complex numbers $y$ and $z$. The first assignment computes the real part of $x$, the second assignment the imaginary part of $x$.

The layout of memory prior to the execution of this example would be
The compiled code for the computation is as follows. (The horizontal lines delineate the groups formed by the T9000 instruction grouper):

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Group</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ldl y</td>
<td>Group 1:</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>fpdlnlsn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ldl z</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>fpdlnlmlsn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ldl y</td>
<td>Group 2:</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>ldnlp 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>fpdlnlsn</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>ldl z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>ldnlp 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>fpdlnlmlsn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>fpsub</td>
<td>Group 3:</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>ldl x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>fpstnlnsn</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>ldl y</td>
<td>Group 4:</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>fpdlnlsn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>ldl z</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>17</td>
<td>ldnlp 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>fpdlnlmlsn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>ldl y</td>
<td>Group 5:</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>ldnlp 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>fpdlnlsn</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>22</td>
<td>ldl z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>fpdlnlmlsn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>fpadd</td>
<td>Group 6:</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>ldl x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>ldnlp 1</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>27</td>
<td>fpstnlnsn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We assume that at the start of the code sequence three words of workspace have been reserved to hold pointers to the base of $x$, $y$, and $z$. The first 13 instructions calculate the real part of the result, the second 14 calculate the imaginary part.
The code operates as follows:

<table>
<thead>
<tr>
<th>Description</th>
<th>State after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $(y-&gt;re)$ is pushed onto the stack</td>
<td>Areg = $(y-&gt;re)$</td>
</tr>
<tr>
<td>2 $y-&gt;re$ is pushed onto the fp stack</td>
<td>FP$\text{Areg} = $y-&gt;re$</td>
</tr>
<tr>
<td>3 $(z-&gt;re)$ is pushed onto the stack</td>
<td>Areg = $(z-&gt;re)$</td>
</tr>
<tr>
<td></td>
<td>FP$\text{Areg} = $y-&gt;re$</td>
</tr>
<tr>
<td>4 $z-&gt;re$ is pushed onto the fp stack</td>
<td>FP$\text{Areg} = $z-&gt;re$</td>
</tr>
<tr>
<td>and the product is formed</td>
<td>FP$\text{Areg} = $z-&gt;re * $y-&gt;re$</td>
</tr>
<tr>
<td>5 $y$ is pushed onto the stack.</td>
<td>Areg = $y$</td>
</tr>
<tr>
<td></td>
<td>FP$\text{Areg} = $z-&gt;re * $y-&gt;re$</td>
</tr>
<tr>
<td>6 Areg is incremented by 1 word</td>
<td>Areg = $(y-&gt;im)$</td>
</tr>
<tr>
<td>generating $(y-&gt;im)$</td>
<td>FP$\text{Areg} = $z-&gt;re * $y-&gt;re$</td>
</tr>
<tr>
<td>7 $y-&gt;im$ is pushed onto the fp stack</td>
<td>FP$\text{Areg} = $y-&gt;im$</td>
</tr>
<tr>
<td></td>
<td>FP$\text{Breg} = $z-&gt;re * $y-&gt;re$</td>
</tr>
<tr>
<td>8-10 $z-&gt;im$ is pushed onto the fp stack</td>
<td>FP$\text{Areg} = $z-&gt;im$</td>
</tr>
<tr>
<td>then $(z-&gt;im) * (y-&gt;im)$ is formed</td>
<td>FP$\text{Areg} = $z-&gt;im * $y-&gt;im$</td>
</tr>
<tr>
<td>11 The value of the real part of the product</td>
<td>FP$\text{Areg} = $z-&gt;re * $y-&gt;re$ - $z-&gt;im * $y-&gt;im$</td>
</tr>
<tr>
<td>($x-&gt;re$) is computed</td>
<td></td>
</tr>
<tr>
<td>12-13 FP$\text{Areg}$ is stored into $x-&gt;re$</td>
<td></td>
</tr>
<tr>
<td>14-27 $x-&gt;im$ is computed and stored</td>
<td></td>
</tr>
</tbody>
</table>

7.1 Execution of groups

We will now look at how this sequence will execute on the T9000 pipeline. We assume that the pipeline is already full when the sequence starts executing.

The diagram below shows on a cycle-by-cycle basis how the groups proceed through the pipeline:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>G1</td>
<td>G2</td>
<td>G3</td>
<td>G4</td>
<td>G5</td>
<td>G6</td>
<td>G7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local</td>
<td></td>
<td>G1</td>
<td>G2</td>
<td>G3</td>
<td>G4</td>
<td>G5</td>
<td>G6</td>
<td>G7</td>
<td>G8</td>
<td>G9</td>
<td>G10</td>
<td>G11</td>
</tr>
<tr>
<td>Address</td>
<td>...</td>
<td>...</td>
<td>G1</td>
<td>G2</td>
<td>G3</td>
<td>G4</td>
<td>G5</td>
<td>G6</td>
<td>G7</td>
<td>G8</td>
<td>G9</td>
<td>G10</td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
<td></td>
<td>G1</td>
<td>G2</td>
<td>G3</td>
<td>G4</td>
<td>G5</td>
<td>G6</td>
<td>G7</td>
<td>G8</td>
<td>G9</td>
</tr>
<tr>
<td>ALU/FPU</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>G1</td>
<td>G2</td>
<td>G3</td>
<td>G4</td>
<td>G5</td>
<td>G6</td>
</tr>
<tr>
<td>Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G1</td>
<td>G2</td>
<td>G3</td>
<td>G4</td>
<td>G5</td>
</tr>
</tbody>
</table>

It is instructive to look at how a couple of groups flow through the pipeline.
7.1.1 Group 1
This consists of \textit{ldl y; fp}ldnlsn; \textit{ldl z; fp}ldnlmulsn and is executed over cyles 2 through 7.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5 and 6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{ldl y}</td>
<td>local</td>
<td></td>
<td></td>
<td>non-local</td>
<td></td>
</tr>
<tr>
<td>\textit{fp}ldnlsn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{ldl z}</td>
<td>local</td>
<td></td>
<td></td>
<td>non-local</td>
<td>ALU/FPU</td>
</tr>
<tr>
<td>\textit{fp}ldnlmulsn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On cycle 2 the two \textit{ldl} instructions are executed in the local stage, on cycle 3 the group occupies the address stage, but no operation is performed, on cycle 4 the two loads from memory are performed in the non-local stage, the floating point multiply is performed on cycles 5 and 6 in the ALU/FPU stage, and finally, on cycle 7 the group occupies the write stage but performs no action. Note that on cycle 8 the write stage is empty, Group 1 has completed its operation but Group 2 is not yet ready to enter this stage. The T9000 processor pipeline allows empty stages ("bubbles") to be created and to be squeezed out.

7.1.2 Group 2
This consists of \textit{ldl y; ldnlp 1; fp}ldnlsn; \textit{ldl z; ldnlp 1; fp}ldnlmulsn and is executed over cyles 3 through 9.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>3</th>
<th>4</th>
<th>5 and 6</th>
<th>7 and 8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{ldl y}</td>
<td>local</td>
<td></td>
<td></td>
<td>non-local</td>
<td></td>
</tr>
<tr>
<td>\textit{ldnlp 1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{fp}ldnlsn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{ldl z}</td>
<td>local</td>
<td></td>
<td></td>
<td>non-local</td>
<td>ALU/FPU</td>
</tr>
<tr>
<td>\textit{ldnlp 1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{fp}ldnlmulsn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The execution of this group is similar to Group 1. However, it should be noted that for Group 2 the cycle spent in the address stage is used to execute the 2 \textit{ldnlp} instructions. Also of note is that 2 cycles are spent in the non-local stage. This is not because they are inherently needed, but because the previous group occupies the following stage for 2 cycles. Thus even if there is a memory bank conflict when accessing \textit{y->im} and \textit{z->im}, and it takes 2 cycles to access both values, the overall speed of the pipeline is not slowed.

7.2 Pipeline activity cycle-by-cycle
We can also examine the behaviour of the pipeline in terms of what each stage is doing on any particular cycle.

7.2.1 Cycles 5 and 6
The diagram below shows the activity in each stage of the pipeline on cycles 5 and 6.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Fetch}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{Decode}</td>
<td>G5 \textit{decoding}</td>
<td>G5 \textit{decoding}</td>
</tr>
<tr>
<td>\textit{Local}</td>
<td>G4 \textit{ldl y, ldl z}</td>
<td>G4 \textit{ldl y, ldl z}</td>
</tr>
<tr>
<td>\textit{Address}</td>
<td>G3</td>
<td>G3</td>
</tr>
<tr>
<td>\textit{Cache}</td>
<td>G2 \textit{fp}ldnlsn \textit{fp}ldnlmulsn</td>
<td>G2 \textit{fp}ldnlsn \textit{fp}ldnlmulsn</td>
</tr>
<tr>
<td>\textit{ALU/FPU}</td>
<td>G1 \textit{fp}ldnlmulsn</td>
<td>G1 \textit{fp}ldnlmulsn</td>
</tr>
<tr>
<td>\textit{Write}</td>
<td>...</td>
<td>empty</td>
</tr>
</tbody>
</table>
Apart from the write stage which has emptied by cycle 6, each stage of the pipeline is occupied by the same group on both cycles. This is determined by G1 requiring 2 cycles to perform its floating point multiply in the ALU/FPU stage.

7.2.2 Cycle 11

The diagram below shows the activity in each stage of the pipeline on cycles 11.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>... decoding</td>
</tr>
<tr>
<td>Local</td>
<td>...</td>
</tr>
<tr>
<td>Address</td>
<td>G6 ldnip 1</td>
</tr>
<tr>
<td>Cache</td>
<td>G5 fpldnsn fpldnimulsn</td>
</tr>
<tr>
<td>ALU/FPU</td>
<td>G4 fpldnimulsn</td>
</tr>
<tr>
<td>Write</td>
<td>G3 fpstnlsn</td>
</tr>
</tbody>
</table>

This demonstrates that although a group may contain only two references to memory, the pipeline may make three access to memory in a single cycle. Here the accesses are being made in the non-local stage by G5 and in the write stage by G3.

7.3 Pipeline parallelism and execution rate

As the above illustration has shown, the T9000 processor uses pipeline parallelism in a fat (multiple execution unit) pipeline to achieve fast execution of the transputer instruction set. The maximum size of group the pipeline can accept is determined by the instruction fetcher and grouper examining up to 8 bytes at once. The pipeline can accept 1 group per cycle (peak), which gives a maximum execution rate of 8 instructions per cycle. However, as the T9000 can fetch only 4 bytes per cycle, that the peak sustainable instruction execution rate of the T9000 is only 4 instructions per cycle (200 MIPS).

In the above example the execution rate is determined by the speed of the FPU, rather than the instruction fetch rate. In this case the FPU executes groups 1, 2 and 3 over cycles 5 through 10. That is:

- 6 cycles for 3 groups
  - for 13 instructions gives > 2 instructions/cycle
  - gives about 108 MIPS
  - for 3 fp operations gives 25 MFLOPS
  - for 19 bytes of code gives 3.17 bytes/cycle

In practice, this code is unlikely to perform as well as these figures suggest. It is quite possible that there will be bank conflicts between the instruction fetcher and the data accesses which may slow things down. The example also illustrates the benefit gained from the fpldnisnmul instruction which decreases the quantity of code to execute from 23 bytes to 19 bytes.

8 Summary

The T9000 transputer has high performance processor. This has been achieved by extensive use of caching and a novel processor implementation. The processor uses a fat pipeline and dispatches several dependent instructions into the pipeline each cycle. The resulting processor is able to saturate a 25 MFLOP floating point unit.
REFERENCES


Next Generation Transputers and Beyond – 3

Towards General Purpose Parallel Computers

Roger Shepherd, David May and Catherine Barnaby
INMOS Limited, Bristol, UK

1 Introduction
Over the last decade, several different parallel computers have been developed; these
have been used in a wide range of applications. Increasing levels of component integration,
coupled with difficulties in further increasing clock speed, make parallel processing tech-
nically attractive. By the late 1990s, chips with $10^8$ transistors will be in use, but design
and production will continue to be most effective when applied to volume manufacture.
A universal parallel architecture would allow cheap, standard multiprocessors to become
pervasive, in much the same way that the Von Neumann architecture has allowed standard
uniprocessors to take over from specialised electronics in many application areas.

1.0.1 Scalable performance
One of the major challenges for universal parallel architecture is to allow perform-
ance to scale with the number of processors. There are obvious limits to scalability:

- For a given problem size, there will be a limit to the number of processors which can
  be used efficiently. However, we would expect it to be easy to increase the problem
  size to exploit more processors.

- There will in practice be technological limits to the number of processors used. These
  will include physical size, power consumption, reliability and thermal density. How-
  ever, as we expect performance/chip to achieve $10^8-10^9$ flops during the 1990s, the
  most significant markets will be served by machines with up to $10^3$ processors.

1.0.2 Software portability
Another major challenge for a universal parallel architecture is to eliminate the
need to design algorithms to match the details of specific machines. Algorithms must be
based on features common to a large number of machines, and which can be expected to
remain common to a large number of machines as technology evolves. Both programmer
and computer designer have much to gain from identifying the essential features of a
universal parallel architecture:

- the programmer because his programs will work on a variety of machines - and will
  continue to work on future machines.

- the computer designer because he will be able to introduce new designs which make
  best use of technology to increase performance on the software already in use.

2 Universal Message Passing Machines
A universal message passing machine consists of:

- $p$ processing nodes with concurrent processing and communication (and preferably
  process scheduling).

- interconnection networks with scalable throughput (linear in $p$) and bounded delay
  (expected $\log(p)$).

Programs for message passing machines normally consist of a collection of concurren-
t processes which compute values and periodically communicate with each other. These
programs must take into account the relationship between the communication throughput and the computation throughput of the message passing machine. We will call this ratio the grain \( g \) of the architecture, and measure it as operations/operand. For simplicity, we will assume that a processor performs an operation in one clock tick, so that we can measure the grain in ticks/operand. In practice, we would like to fix the grain to about \( g = 1 \); this is entirely consistent with the rate at which operands are moved between processor and memory in unprocessors. For universal message passing machines, we will require that this ratio is maintained for system-wide inter-processor communication. It turns out that this can be achieved using 1990s VLSI technology.

Ideally, we would like to avoid the need for the programmer to be concerned with \( g \), with programs being written at a higher level using, for example,
- Array manipulation
- Big DO-PARs
- Explicit parallelism with lots of small processes

The compiler will then take as parameter the grain \( g \), and will parallelise or serialise so as to express a program as a collection of \( v \) virtual processors (processes) of grain \( > g \) and cycle-time \( c \) (ticks). We assume that the processes are cyclical, and in each cycle perform \( c/g \) communications and \( c \) operations. Notice that we want to keep the grain of the software as low as possible so as to exploit all possible parallelism for a given problem size, but at least \( g \) to avoid processor idling.

The output of the compiler is a program suitable for use on all universal machines of grain \( g \). (This is why \( g \) plays no more part in our discussion of the construction of a universal message passing machine). We expect to keep the program in this form, and perhaps distribute it in this form. We note that \( g \) is fixed for a range of machines based on the same components, and further that there is likely to be little variation in \( g \) even for machines based on different components. This means that the compiled program is likely to be re-useable.

To load a compiled program for execution, we make use of a loader which takes as parameter the latency of communication: \( l \) (ticks). This will vary from machine to machine and will scale as \( \log(p) \) for realisable networks. The loader will allocate at least \( l/c \) virtual processors to at most \((v \times c)/l \) processors. There would be no point in attempting to use more processors than this, as this would result in the processors idling some of the time; it would be better instead to leave some processors available for allocation to another program. Thus the program will run at optimal efficiency on a \( p \)-processor machine provided that \((v \times c) > (l \times p) \).

Notice that our loader ensures that there will always be enough processes on each processor to ensure that (at least) one is executable; the others will be waiting for communication to complete. This means that we will need to use at least \( \log(p) \) more processes than processors. Another way to think of this is that we could use a specialised machine exactly matched to the algorithm in which each processor executes only one process; this would offer \( \log(p) \) more performance. Specialised parallel computers will still be needed for maximising performance where the problem size is limited!

We note that our proposal for universal message passing is closely related to Valiant's proposal for Universal PRAMs [1] in which \( l = \log(p) \) and \( c = 1 \).

3 Processing Components For Universal Machines

VLSI technology enables a complete computer to be constructed on a single silicon chip. The T9000 transputer [2] contains an integer processor and a scalar floating point unit; peak performance is 200 MIPS and 25 (scalar) MFLOPS. Each communication link provides simultaneous transfer of data in both directions at about 10Mbytes/second. Thus the T9000 can supply data from its four links at about 40Mbytes/second, whilst simultaneously receiving data at 40Mbytes/second. Internally, the T9000 is designed to maintain high communication rates whilst also processing data supporting a grain \( g = 1 \).
for single length floating point programs communicating through all four links.

Transputers have the ability to communicate directly with each other via their communication links; this enables transputers to be connected together to construct multiprocessor systems to tackle specific problems. For the construction of universal machines, the links will be used to connect the transputers into a communications network constructed from communication components.

Each transputer includes a hardware kernel with the ability to execute many software processes at the same time, to create new processes rapidly, and to perform communication between processes within a transputer and between processes in different transputers. All of these capabilities are integrated into the hardware of the transputers, and are very efficient. Kernel operations (process scheduling, communication) take about 1 microsecond, in contrast to the 100 microsecond overheads typical of software kernels. The close integration of the processor and links within the T9000 transputers provides high-throughput, low-delay interprocess communication via the links, with hardware being used to multiplex many interprocess communication channels along a single physical link.

4 Communication Components For Universal Machines

During the 1990s, VLSI will be applied to the development of components for communication networks. One example is the the INMOS C104 packet routing chip [2], designed to connect 32 links of the kind used on the T9000 transputer. The C104 contains a 32-way crossbar switch, in which all of the 32 inputs can be routed simultaneously to the 32 outputs. Routing delays are minimised by the use of wormhole routing, in which a packet can start to be output from a switch whilst it is still being input.

A packet arriving at a switch is routed according to its header. If the required outbound link is available, the packet utilises the link. However, if the link required is currently in use, the packet will be blocked. The tail of the packet may now start to catch up with the head. If there is enough buffering, the whole packet may be taken into a buffer, waiting to have access to its required output. Therefore if the network is very busy, the performance will approximate to the performance of a store-and-forward network. The C104 provides one packet of inbound buffering, and one packet of outbound buffering on each link.

The C104 supports universal routing [3]. Universal routing requires a packet to be sent to a randomly chosen intermediate node before it travels to its real destination. Any of the links on the C104 can be set to create a random header for each inbound packet on that link. At the randomly chosen intermediate node, this random header is deleted, leaving the original header to route the message to its real destination.

All routing, header creation and deletion is performed on a per link basis. There is no shared resource within the C104. This has the effect of making the links of the network the shared resources, rather than the nodes of the network.

A full description of the C104, and of its use in networks is given in [4].

5 Networks For Universal Message Passing Machines

In a communication network connecting \( p \) terminals, we can realistically expect the distance a packet will travel to increase with \( \log(p) \). Consequently, if we wish to maintain throughput per terminal, the number of packets in flight from each terminal will scale with \( \log(p) \). Therefore network capacity required for each terminal will scale with \( \log(p) \). The total capacity of a network with \( p \) terminals must therefore scale as \( p \times \log(p) \). One structure which achieves this is the hypercube or binary n-cube. Another structure is the (indirect) butterfly network, which has constant degree. Conversely, the two-dimensional grid does not maintain throughput per node as the network scales.
5.1 Network Performance

In order to demonstrate that devices such as the C104 can support the communication requirements of universal parallel computers, we summarise recent simulation results. A more complete study can be found in [5].

The simulations examine hypercubes and two-dimensional grids; for both of these, three network sizes are measured for three traffic patterns. The simplest of traffic pattern, random traffic, operates by continuously supplying packets at every node of the network. The destination of this packet is chosen at random from all possible destination addresses. This pattern will overstate the throughput achievable in general as it distributes the traffic over the network in a manner similar to universal routing.

A more interesting pattern is systematic traffic, in which each source sends all of its packets to a specific destination. Each of the patterns chosen is a permutation, so that no two source nodes send to the same destination. For each network, a systematic traffic pattern is chosen. These patterns represent operations which we could reasonably expect an algorithm to perform and we would require our bnetwork to support them efficiently. However, in each case these patterns create severe hotspots in the network.

Finally, we show the effect of universal routing on the systematic traffic.

Figure 1 illustrates that on the hypercube, throughput scales as predicted for random traffic; that is the throughput/node remains constant. On the other hand, the systematic permutation shows a large decrease in throughput/node as network size increases. Universal routing has the effect of adjusting bringing the performance of the systematic traffic towards that of the random traffic. The small variations of throughput with network size are attributable to properties of the random number generator.

![Figure 1: Throughput/node varying with network size on a hypercube](image)

For contrast, Figure 2 shows the throughput of a two-dimensional grid. In this case, the random traffic shows that throughput/node degrades with increasing network size. This is to be expected, as the grid does not increase network capacity at a suitable rate.
Systematic traffic shows that the throughput on a grid can both be affected considerably by the traffic pattern and, as with the hypercube, the throughput/node decreases with the network size. Universal routing again pulls the behaviour back towards the random traffic, providing similar scalability in throughput which is now limited only by the overall capacity of the network.

![Graph showing throughput vs. number of nodes](image)

Figure 2: Throughput/node varying with network size on a grid

5.2 Performance Predictability

In the previous section we have shown that universal routing on the hypercube will provide the throughput need to implement a universal message passing machine. This section we will show that universal routing also improves the predictability of the network.

We will consider a class of permutations of the hypercube known as the perfect shuffle. As in any permutation, each node in the network sends data to a distinct destination node; in the perfect shuffle the destination node number is obtained from the source node by rotating the bits representing the source node number by a particular amount. A rotation of 1 gives a 2-way shuffle, a rotation of 2 gives a 4-way shuffle, and so on. The perfect shuffle is the basis of many fast parallel sorting algorithms.

Figure 3 shows the time required to perform perfect shuffles on an 8-dimensional hypercube. For each shuffle, every node sends 20 packets to its destination, and the time taken for all packets to reach their destination is measured. Figure 3 shows the time taken for rotations of from 0 to 8 using deterministic routing. Also shown is the time taken for a 0 rotation shuffle under universal routing.

This shows that the deterministic routing gives a wide variation in run-time. For instance, changing a program to be an 8-way shuffle rather than a 4-way shuffle would increase the network delivery time by a factor of 2. With universal routing the time taken remains approximately constant. This is a major advantage, since calculating a bound on the run-time requires that the worst case must be taken into account.
6 Is All This Reasonable?

To show that the programming process outlined earlier can be applied to realistic problems for realisable machines, we consider a simple image processing example, that of image smoothing.

We assume that each transputer is connected to a hypercube network by a single link. Our simulations have shown that hypercubes operate at about 80% of peak throughput/link. (For a T9000 this corresponds to 8 Mbyte/second). This gives a grain $g$ of about 5, assuming that we can sustain about 10 Mflops on the image processing operation. For image smoothing, which consists of 5 operations per point (4 additions and one division), we can split a picture into 4 pixel by 4 pixel squares. This provides $16 \times 5$ floating point operations and 16 communications - a grain of exactly 5 and a cycle of 80.

We know that for a 64 processor machines, the probability of any message being delayed by more than 800 is very small ($< 10^{-3}$). So if we load at least 10 processes onto each processor, we can be sure that the machine will run at high efficiency, achieving $64 \times 10$ Mflops on a $10 \times 64 \times 16$ point image. This is 640 Mflops on an image of 10240 points. As this is well below the size of typical images, we might want to scale up the problem size and/or the machine. This is possible without modifying the algorithm or adjusting its grain: all we need to do is to add more processes, and rely on the loader to load sufficient processes/processor.

7 Summary

We have used the following result from contemporary computer science:
- the ability of two-phase randomised routing to support scalable throughput and low delay when routing on hypercubes (even when routing among the $p \log(p)$ processes distributed among $p$ processors)
  together with the existence of message-passing hardware:
processors, preferably with efficient process scheduling, in which processing throughput and communication throughput is balanced

communication networks with scalable throughput and low delay

and have shown that we can already construct scalable universal message passing machines. For these machines, we can write scalable, portable software exploiting message passing.

REFERENCES


A PORTABLE UNIFORM INTERFACE TO REAL-TIME OPERATING SYSTEMS

Robert D. Russell

Computer Science Department,
University of New Hampshire, Durham, USA rdr@cs.unh.edu

Abstract

This paper discusses the requirements and problems associated with providing portable real-time software. A proposed solution to the difficulties is vos, a Virtual Operating System, designed and implemented at CERN. Vos enables identical software to run on many different host systems, including UNIX, VMS, and OS-9 platforms. It is based on POSIX and includes the implementation of some real-time extensions proposed for POSIX. Examples of the features of vos and its use in applications are given, along with measurements of its performance and details about its implementation.

1. BACKGROUND

During the decade of the 1980's, one of the most popular computers for real-time data acquisition in high energy physics was Digital Equipment Corporation's VAX computer running the VMS Operating System [1]. There were many reasons for its popularity:

(1) It was the historical successor to the PDP-11, one of the most popular computers for data acquisition in the 1970's.

(2) The architecture of the VAX, along with the interfaces provided for it, were well suited to the physics environment.

(3) The hardware was available across a wide performance range, from small VAXstation 2000's and 11/730's to large 6500's, and in a wide variety of configurations, from "micros" to "minis" to "workstations" to "almost mainframes".

(4) The VMS operating system provided the real-time features needed to read-out data, record it to tape, filter it, and sample it under programmer control.

(5) The system provided good support for several popular languages, among them Fortran-77, Pascal, ADA, and Assembler.

(6) The programming languages were all extended to allow direct access to operating system features, and components written in different languages could easily be integrated together into a single application.

(7) The system provided interactive time-sharing services and a powerful command language interpreter that worked together with the real-time features in one CPU.

(8) The system supported extensive networking facilities in the form of DECNET.

The equipment, operating system, extended languages, and networking were all proprietary items, sold and supported by DEC but not by any other vendor. However, because the support and documentation from DEC was good, the high energy physics community was able to survive and even prosper in this closed environment, and a lot of software was developed for it.
At the end of the 1980's, competitors were marketing a number of attractive workstations with better price-performance ratios based on RISC technology, the UNIX operating system, and the Internet networking protocols. Even Digital Equipment itself introduced a RISC-based workstation running ULTRIX [2,3], DEC's version of Berkeley UNIX, and providing networking services based on both the Internet TCP/IP protocol and the DECNET protocol. The obvious speed and price advantages of these systems over the VAX compelled researchers to turn to them. However, a big problem was the basic incompatibility between VMS and ULTRIX/UNIX. None of the software written for VAX/VMS would run on RISC/ULTRIX. The command language interpreter was different, the languages were different, VMS extensions to common languages were not supported, the operating system calls were all different, and even the networking was different, since most existing UNIX networking software was based on the Internet TCP/IP protocol. What was to be done with the huge investment in all the software written for VAX/VMS? Should it be rewritten from scratch? Should it be "hacked" to run under ULTRIX?

Because of the sheer amount of software, it was clearly impractical to rewrite it all from scratch. There simply wasn't the time nor manpower to do this. Furthermore, it was not necessary for large chunks of the code, since these dealt with internal data structures and computations that were really independent of the details of the real-time environment. There was, however, an additional problem of integrating any new developments with what already existed. In particular, most new workstations would be expected to operate together with existing VAX systems on the same experiment. It would therefore be ideal if the existing software could be modified in such a way that it would run, at the very least, on both VAX/VMS and RISC/ULTRIX systems, since having two slightly different versions of the same programs causes nightmares for system managers, programmers, and users.

2. PORTABILITY

The simple answer to these difficulties is "portable software", although the realization of this answer is admittedly not so simple. Software is "portable" when it can be moved from one computer system to another with very little change. It is hard to even quantify how much change is "very little", but the effort involved must be substantially less, by at least several orders of magnitude, than the effort invested in the original software. Another important requirement of portable software is that after a port, there must be one "master" version of the software that can easily be maintained for all systems to which it has been ported. Versions for different systems that "drift apart" after the initial port become different pieces of software, and the "portability" is lost completely.

There are several aspects of portability that need to be considered, since not all portable software satisfies all of these aspects:

(1) Portability of the source code of the application.
(2) Portability of whatever code or commands users of the application write.
(3) Portability of the users themselves between different hosts running the ported software.
(4) Integration of the software into the style and conventions of the different hosts.

2.1. Source Code Portability

The first aspect, "portability of the source code of the application", is what most people think of when they hear the term "portable software". This means that the source files can be recompiled, linked, and run on a new host with few, if any, changes to the source, certainly not massive reprogramming. One of the key developments tending to improve the
chances that source code will be portable is the emergence of standards for programming languages. The Fortran-77 Standard [4] was one of the first to be developed, in 1978, but many more languages were standardized during the 1980's: ADA [5] was standardized in 1980, Pascal [6] was standardized in 1981, and C [7] was (finally) standardized in 1989. The idea of a Language Standard is to define what facilities a programmer in that language can expect to find on all implementations of that language, so that, by confining himself or herself to exactly the facilities defined in the Standard, a programmer can reasonably hope that his program will compile and run identically, without change, on all hosts supporting that language.

The difficulty with many Standards is that they tend toward the "least common denominator" of existing implementations. This means that desirable features are often left out, or are relegated to the all inclusive category of "implementation defined", which, of course, totally destroys the goal of portability. The Standards have tended to be weakest in their definition of language interaction with the operating system environment, exactly the area of most concern to real-time programmers! For example, the following is a partial list of problems with Standard Pascal [6] in this area:

(1) There is no standard way to separately compile Pascal programs and then link them together into one executable program. In particular, there is no way to write a run-time library in Standard Pascal.

(2) There is no standard way a Pascal program can call routines written in other languages. Consequently, a Standard Pascal program cannot use any library routines except those defined in the Standard.

(3) There is no standard interface to the external file system, since the interpretation of "program parameters" is implementation defined.

(4) There is no standard way a program can give a name to a file or device in the external file system.

(5) There is no standard way to perform interactive I/O.

(6) There is no standard way for a Pascal program to deal with run-time errors.

2.2. User Command Portability

The second aspect, "portability of whatever code or commands users of the application write", is usually, but not always, the easiest aspect of portability to accomplish. Basically, this aspect deals with the type and form of input expected by the application. For example, if the application is a data base package, this will be the form in which queries are written. Since this form is usually defined by the application itself, it often will be identical on different hosts. Problems arise when different hosts use different character sets, especially when certain characters have special meaning to one host operating system but not to others. Other problems can arise with tape formats, since some hosts may place restrictions on block sizes, labels, naming conventions, etc. The syntax of file names and the way to access different devices, including "temporary" file storage, is almost always a source of difficulties. In general, any "system" format, even something as simple as the current date and time, can cause portability difficulties. One important development in this area has been the emergence of a standardized operating system interface, POSIX [8], which defines some of these formats and naming conventions. We will discuss POSIX more below.
2.3. User Portability

The third aspect, "portability of the users themselves between different hosts", is usually the least considered aspect, and probably the hardest to achieve. It has to do with how easy it is for a user of a software application accustomed to using it on one host to use it on another host. A simple example can illustrate this. Most applications require that the user type input commands or data from the terminal. Suppose the user types something incorrectly and wishes to correct it. What key does he or she type to erase the previous character, to erase the previous "word" (however that is defined) or to erase the complete input line and start over? Suppose the error was simply the omission of a character, word or phrase at some point earlier in the line. How can the user skip back to that point and make the insertion? How can the user even interrogate the system interactively to find out the answers to any of these questions? In general, what kind of intra-line editing is available to a user?

Unfortunately, the answer to all these questions is usually "it depends on the host operating system". Unless the user is familiar with the conventions of each host, he or she is not portable between them, even when running his or her own "portable" application.

2.4. Intra-host Portability

The last aspect, "integration of the application software into the style and conventions of each host", is almost directly in opposition to the previous goal of user portability between hosts. Here what we are talking about is how easy is it for a user familiar with all the conventions of some host system to use an unfamiliar application that has been ported to that host. If the application brings with it the "style" of its original host, the answer may be "not at all easy". Again a simple example can serve to illustrate this point. On some systems the convention has grown up to "prompt" for almost everything, with or without meaningful explanations and/or "help" facilities. On other systems, prompts are virtually non-existent. An application written to repeatedly prompt for everything will not be appreciated by users on "silent" hosts, and applications that do not prompt for anything will be unusable by users on hosts that expect to be prompted for everything. Similar considerations apply to the form and order of input parameters (i.e., in a copy, is it "old" followed by "new", or "new" followed by "old"?) Even worse are the differences between a "command line oriented" host, and a "menu oriented" host, since applications are rarely written to accommodate both approaches, and yet the styles are so vastly different that one is completely unusable in the context of the other.

Another consideration in the usefulness of a ported program is how well it integrates into the conventions of the new host. In particular, many systems establish conventions for use of default I/O devices, for use of temporary files, for logging reports and errors, etc. Part of porting a program is tailoring it to fit these new conventions, because if this is not done, it will not be accepted by the general user community on the new host, in spite of its other merits. For example, the formats it uses for input and output files must be compatible with other, more general utilities on the host to copy, edit, and print files.

3. MOTIVATION

During the 1980's, the Online Computing Group of the DD Division at CERN developed a suite of modular data acquisition programs called "MODEL" [9,10]. The goal was to provide basic building blocks from which a physics experimenter could construct a data acquisition system suited to his or her particular needs. Each building block was to be a self contained module that was independent of the other modules yet would work together...
with other modules as required. The software was constructed specifically for VAX/VMS systems, and was written in several different languages (Fortran, Pascal, and ADA). In 1989 an immediate task was to port some components of the MODEL suite to a DECstation running ULTRIX for use in an experiment during the spring of 1990. As indicated in the background section above, it was felt that the proper approach was to reimplement those parts of MODEL that depended directly on VAX/VMS so that they would run on both VAX/VMS and RISC/ULTRIX at the very least, with the longer term goal of making them portable to other UNIX-based workstations and even to computers running entirely different operating systems, such as OS-9. The new, portable versions were expected to replace the existing VMS-based versions as well as to be used in the UNIX-based environment so that we would not end up with two similar versions of each MODEL module that would have to be maintained and upgraded separately.

A second motivation for adopting a portable approach to MODEL software was an attempt to integrate the historically divergent development being undertaken on the small Motorola 68000 based systems. These systems, such as the VALET+ [11], are dedicated systems built around a 68000/VME configuration. They are used for first-level readout, for testing and diagnostics, and for other dedicated, front-end type tasks. They are often embedded into a larger system controlled by a VAX system running MODEL components. The software for these systems consists of an interactive PILS interpreter and compiler, a RPC network interface, a number of CERN libraries for histogramming, graphics, FASTBUS, and CAMAC, and SPIDER [12], a simple data acquisition system. Although this software was developed separately from and independently of MODEL, a lot of the requirements were the same and there was a need for better integration of the software so that, for example, parts of it could be moved from a back-end VAX into a front-end VALET+, or vice versa, in response to changing demands of the experimental configurations. Although no further development of the VALET+ was foreseen, it was clear that a successor system would be necessary, and it was also clear that the design and implementation of this system should be as portable as possible so that the software would not be locked into a particular hardware, since the improvements in chip technology were accelerating at such a pace as to make it impossible to consider standardizing on a single chip. Furthermore, although it seemed as if the immediate successor would be based on OS-9 [13], it was not at all clear that this operating system would be suitable over the longer term, and it was therefore important not to get locked into a dead-end development of software that would be unusable by the successor to OS-9.

These two motivations, the immediate need to move MODEL to the RISC/ULTRIX world, and the longer term need to provide portable software as a way of unifying divergent software developments and ensuring the future utility of new developments, lead to the idea of "VOS” [14], a “Virtual Operating System”. Vos is not a separate, stand-alone operating system. Rather, it is a uniform interface between real-time applications and host operating systems that is designed to isolate an application from the peculiarities of a host and thereby make it portable to several hosts. Vos itself has to be reimplemented for each new host, but once that is done, all applications using vos can be moved to the new host without any modification.

The main design objectives for vos were:

(1) To provide a uniform operating system interface by which applications using it would be portable across several families of host operating systems.

(2) To provide real-time features in an efficient manner.
(3) To provide network access in a protocol-independent manner.
(4) To provide "natural" programming interfaces in several programming languages.
(5) To avoid specifying or implementing more than was necessary.
(6) To implement everything "on top of" existing operating systems, requiring no changes or modifications to a host system in order to accommodate vos.

The initial implementation targeted three "families" of operating systems: VMS, OS-9, and the sundry variants of UNIX. The initial networking bases were DECNET and the Internet TCP/IP. The initial programming languages were C and Fortran.

Defining and implementing an interface such as vos is not something to be undertaken lightly. Before deciding to go this route, we looked into a number of other alternatives that should be mentioned briefly here. The first of these was "ORKID" [15], an "Open Real-time Kernel Interface Definition". This is a standard developed by a working group sponsored by VITA, the VMEbus International Trade Association, to allow "users to create robust and portable code". It is restricted to real-time, kernel-level functionality. It existed only in a draft form in 1989, when vos was undertaken, and no implementations were available to us, nor did it appear likely that any would be available in the foreseeable future. It was also unclear how widely accepted this standard would be, since it had very limited corporate sponsorship, and the large computer manufacturers were noticeably absent from the list of supporters. For our immediate needs, we did not require all the features ORKID specified, and yet it lacked some of the features we did require, such as networking, error reporting, interaction with the command interpreter, and bindings to several languages. Furthermore, ORKID was not part of a larger system that included a file system, command interpreter, program development tools, etc. This means that all development would have to be done on a different system, and that application programs needing access to a file system, for example, as well as to real-time facilities, would have to be developed outside the context of ORKID, at least in part. We therefore used ORKID only as a source of reference, but did not attempt to consider any sort of compliance with its definition.

A second alternative that we looked into was a number of "Real-time UNIX" systems. These are all proprietary systems that have extensively modified an existing UNIX system to incorporate various real-time features. Many of these work by utilizing two processors: the first for program development in a conventional UNIX environment, the second a stand-alone processor into which the real-time application is loaded and run. The second processor often contains only a stripped-down kernel that may or may not resemble UNIX, and if it is available at all, file system access is usually provided via interaction with the front-end development processor. We tended to discount these systems because of their proprietary nature and the historical aversion which many physicists feel for downline loading into computers over which they have only indirect control.

An alternative that we did seriously consider was "POSIX", a "Portable Operating System Interface (based on uniX)". This is a standard developed and approved by IEEE, the International Organization for Standardization (ISO), and the International Electrotechnical Commission (IEC). Actually, it is several standards, only two of which have been officially approved. The following is a list of the first ten components of this standard, with their status as of July, 1991.

(1) POSIX 1003.1 — System Application Program Interface (API), C Language [8]. This Standard defines a set of system calls, data structures, data types, definition files, and other conventions that together provide a standardized interface which a C program can use to access some basic operating system facilities. These facilities include process
creation, deletion and control, some forms of interprocess communication and signaling, file system primitives, interactive terminal control, basic timer operations, and limited interaction with the command interpreter. The first version of this standard was approved in 1988, a revised version in 1990, and virtually all major computer manufacturers have committed themselves to providing a POSIX compliant environment in the near future. Although initially defined only for the C language, efforts are underway to develop "Language Bindings" to ADA and Fortran as well.

(2) POSIX 1003.2 — Shell and Utilities. The definition of the interactive command language and the common commands used by it. This is based on the historical UNIX Shell and its utilities. It is currently in Draft 10, published in July 1990.

(3) POSIX 1003.3 — Test Methods. Already Standardized.

(4) POSIX 1003.4 — Real-time Extension for Portable Operating Systems [16]. This Standard defines a number of modifications and additions to POSIX 1003.1 to allow C programmers to write real-time applications, where real-time is defined as: "the ability of the operating system to provide a required level of service in a bounded response time". The facilities include binary semaphores, shared memory, priority scheduling, asynchronous event notification, high resolution timers, deterministic interprocess communication, synchronous and asynchronous I/O, and real-time files. The latest draft, Draft 10, was published in January 1991. As with POSIX 1003.1, many of the computer manufacturers are committed, at least verbally, to supporting the POSIX 1003.4 real-time extensions.


(9) POSIX 1003.9 — Fortran Language Bindings. Draft 8, November 1990.


Unlike ORKID, POSIX is a truly international standard, enjoying wide support from the computer industry. In addition to the commitments of the various individual manufacturers, it has also been embraced by both UNIX International and OSF. Of great interest to us was the commitment by DEC to support POSIX 1003.1 and 1003.4 on both their VMS and ULTRIX operating systems.

We therefore decided to base our development as much as possible on POSIX, but this was not without its problems, the biggest of which was with the real-time extensions, POSIX 1003.4 [16]. Unlike POSIX 1003.1, which is essentially a codification of existing UNIX practice, many of the real-time extensions in POSIX 1003.4 are newly designed from scratch by the POSIX committee. This is because historically UNIX has been deficient in the real-time area, and many of the "hacks" introduced to rectify the situation were unsuitable for general application. This has caused some concern in the industry, since a new design always contains elements of uncertainty as far as unforeseen problems and performance are concerned, and this has also prolonged the approval process for this section of the standard. Together these concerns lead to a dilemma: manufacturers are reluctant to implement features that have not been approved and whose specifications may change before being approved, yet until these features are implemented and tested, manufacturers are reluctant to approve new and untried designs. The debate had been underway for several years when we began considering vos, and it continues now, several years later.
There were other problems too. The working committee to consider the networking features of POSIX was just getting started in 1989, and no specifications, even in draft form, were at all imminent. Although POSIX 1003.1 had been first approved in 1988, neither VMS nor ULTRIX were in compliance in 1989. Some components of POSIX, such as the error reporting, were poorly designed. And the only language that could be used with POSIX was C, whereas most of our applications were written in Fortran.

As a result of our deliberations, a strategy emerged that would allow us to get going at once on the task of getting portable software up and running on a variety of different hosts, without having this effort be negated by the eventual appearance of all the POSIX developments. We would define a completely separate interface between the user applications and the operating system. This was called "vos", for "Virtual Operating System", since it was not a "Real" operating system. Although clearly based on POSIX, this interface would not be POSIX. It would provide just those real-time features that our experience dictated would be needed to port all existing MODEL software and to develop new software for our applications. The form of these features would be tempered by whatever restrictions we might encounter in actually implementing this interface on the three widely different operating system families: VMS, OS-9, and UNIX. From the beginning, we would provide a programmatic interface for both C and Fortran, and we would integrate real-time features with networking facilities, error reporting, etc. And, perhaps most importantly, we could provide everything quickly, without endless waiting for committees to deliberate and manufacturers to respond.

Table 1
Why VOS, not POSIX only?

- Natural FORTRAN and C calls
- Network access
- Better error handling
- Portable to non POSIX systems
- Easier integration on present hosts
- Based on HEP-DAQ needs
- Up and running now

So where does POSIX fit into all this? In our implementation. We decided to implement our vos routines in terms of the POSIX features specified in the POSIX 1003.1 Standard and 1003.4 Draft as much as possible. When these features were not implemented on one of our hosts, we would simulate them to the extent necessary to get our routines working. However, if suitable features already existed in the underlying operating system, we would utilize them directly. As discussed later, most of our "simulated POSIX" code runs on top of conventional UNIX code, because UNIX lacks many of the real-time features we needed. VMS, on the other hand, already contained features suitable for direct implementation of most vos facilities without the need to simulate POSIX actions in between.

The advantage of this approach is clear: when and if POSIX real-time extensions are provided by manufacturers, we will simply throw away that part of the vos code that simulates POSIX, giving us a slimmer, faster interface that still retains full compatibility with everything we have done up to that point. If the specifications for some (or all) of the real-time extensions change before they are approved, we have isolated the modifications we will have to make to just the vos interface itself – none of the user application code will have to change. The introduction of POSIX onto different hosts can be gradual, since we can
continue to run our simulation on hosts of manufacturers that are slow to move to POSIX compliance. Finally, the vos interface will provide a convenient mechanism for porting applications to and from systems that never become POSIX compliant.

A schematic diagram of the layers of software is shown in Fig. 1. CATS [17], standing for "Common Access to Transport Service", is a standardized network interface that was easily implemented on top of vos, thereby increasing the ease with which existing code based on MODEL and CATS could be ported via vos.

![Diagram](image)

**Fig. 1** Software layers with VOS and POSIX.

An interesting historical note is that our initial attempt at porting some of the MODEL modules to ULTRIX involved implementing them directly in terms of POSIX components that we simulated on top of ULTRIX. Although we did get one module, OSP [18], working in this manner, we quickly found out the difficulties of this approach. In particular, many of the extensions to POSIX require modifications to existing primitives, such as `open()`, which already exist in UNIX. Clearly we could not modify an existing UNIX which was supported by a manufacturer (we didn’t even have the source code in any case), and we couldn’t redefine the system function without losing access to it (i.e., if we defined our own POSIX `open()`, we could no longer reference the system `open()`, and without this we could not access devices or files). However, without some such modification, a user could not use the extended POSIX specifications directly. Hence the need for a vos layer between the user and the host system.

4. ERROR HANDLING

One of the weaknesses of the POSIX Standard is the way it deals with errors. Essentially, POSIX has adopted the UNIX convention, which was also adopted by the Standard definition of the C Language [7]. This convention defines a global integer variable called `errno`, and requires all system routines to be functions that report errors by storing an error code in `errno` and then returning as the function’s value a designated error flag, which is −1 for many (but not all) functions. The value in `errno` is left undefined if a system routine does not return the error flag value (i.e., `errno` is NOT reset to zero (or any other value) if the function succeeds). The system provides routines to convert an error code to an explanatory string of text, and to print this text onto an error logging device.

There are several major difficulties with this scheme:

1. It is not extensible. All the error codes are predefined in the system. A user cannot define additional sets of error codes and messages and have them integrated with the system error reporting routines.
(2) It is not re-entrant. Since there is only a single global integer containing the most recent error code, any errors generated in an interrupt service routine will overwrite the contents of the global variable, perhaps before the interrupted code has had a chance to utilize the code. To avoid such a "race" condition, each interrupt service routine must begin by saving the value of *errno* locally, and must restore this value just before exiting.

(3) It is not transparent. A user-written library package, for example to build histograms or to interface to CAMAC, must use the same global *errno* as the user is accessing whenever it calls system functions. Therefore, unless each routine in the package saves *errno* on entry and restores it on exit, there may be undesirable interference between errors that are generated internally by the package and those that belong to the user. As already mentioned, the library package cannot in general use *errno* to return its errors, because its own codes and messages will not be understood by the system reporting functions.

In vos we adopted a different approach, one similar to that used by VMS. Most vos routines are defined to be integer functions that return a status value. If the function performed properly, the status value returned is the symbolic constant VOS_NORMAL. If the function detected an error or some other non-normal situation, the status value itself is the error code. The word "error" is a bit too strong here, because the status value may be conveying information that is not really an error. For example, a read operation may return the value VOS_ENDOFFILE, indicating that there is no more data to be transmitted from an input device. This is probably not an error, per se, but rather the normal way of detecting the end of input. For that reason, we will refer to these return values as "status codes". Note that because the return value of the function is used to convey a status value, any "results" produced by the function must be returned via output parameters to the function.

Vos provides several functions for dealing with status codes. The boolean function *vos_success()* accepts a status code as its only parameter and returns true or false, depending on whether the code indicates success or failure. This is often used in a C statement of the form (where *vos_xxxxxx()* is an arbitrary vos function):

```c
status = vos_xxxxxx( ... );
if( vos_success(status) )
    ( /* operation was successful */ )
else
    ( /* operation failed */ )
```

A complementary boolean function, *vos_failure()* accepts a status code and returns true if the code indicates failure, false if it indicates success. Note that VOS_NORMAL is not necessarily the only "success" code, since, as explained below, it is possible to define any number of different success values in order to return additional information about the type of success.

The routine *vos_report_error()* accepts a status code as its only parameter and prints to the standard output unit a text message corresponding to the error, although nothing is printed if the code is VOS_NORMAL. The value returned by *vos_report_error()* is just the value of its parameter (i.e., the status code on which it was asked to report). It can be used in a C statement of the form:
status = vos_xxxxx(...);
if( vos_success(vos_report_error(status)) )
    { /* operation was successful */
    }
else
    { /* operation failed, message already printed */
    }

The routine vos_translate_error() is invoked from C with a statement of the form:

    status = vos_translate_error(status, buffer, max, &length);

where:

status       is the status code returned by a previous vos function.
buffer       is the address of a string into which this routine will put the text of a message explaining the error.
max          is the maximum number of characters that buffer will hold.
&length      is the address of an integer into which this routine will store the number of characters actually written into buffer.

The message string delivered in buffer is the same as the message printed by vos_report_error for the same value of status (assuming buffer is large enough to hold the complete message). If the value of status is VOS_NORMAL, an empty string is returned. The value returned by vos_translate_error() is the value of its status parameter (i.e., the code value it is being asked to translate).

Vos status codes and messages are defined with the aid of a set of tools based on the VAX/VMS error message facility [19]. This enables vos users to define their own status codes and messages, and have them integrated into the vos error reporting. This means that the error handling routines just described will work on user defined status codes and messages in exactly the same manner as they work for the system codes and messages.

Figure 2 shows how genmess is used. The user defines a set of codes and messages in the file "test.msg", which is then input to genmess. A sample input file follows:

```
.TITLE       SAMPLE  a sample input file
.IDENT       "version 1.0.0"
.FACILITY     JUNKY,73/SYSTEM/PREFIX=JUNKY$_

.SEVERITY    SUCCESS
OK            "everything is ok"
GREAT         "everything is great"
SUPER         "this is really super"

.SEVERITY    INFORMATIONAL
TRICKY        "best avoid such tricks"

.SEVERITY    ERROR
BAD           "you blew it"
GROSS         "this is most unfortunate"

.SEVERITY    FATAL
.BASE        64
ALLOVER      "you bought the farm"
UNREAL       "what is your major?"
.END
```
Genmess produces three output files: "test.h", which contains a set of symbolic constants that can be included in any C program that wants to refer to the numeric error codes (such as VOS_ENDOFFILE, for example); "test.c", which contains a set of tables that must be compiled and then linked with any program using these error codes; and "test.lis", which is a listing file suitable for printing. The output file "test.h" produced from the above sample input follows:

```c
/* Include file of error and status messages for C. */
/* Generated by GENMESS V1 on 21-JUL-1991 at 11:10:33. */

/* .TITLE SAMPLE a sample input file */
/* .IDENT *version 1.0.0* */
/* .FACILITY JUNKY,73/SYSTEM/PREFIX=JUNKYS_ */

/* .SEVERITY SUCCESS */
#define JUNKYS_OK 0x00498009 /* "everything is ok" */
#define JUNKYS_GREAT 0x00498011 /* "everything is great" */
#define JUNKYS_SUPER 0x00498019 /* "this is really super" */

/* .SEVERITY INFORMATIONAL */
#define JUNKYS_TRICKY 0x00498023 /* "best avoid such tricks" */

/* .SEVERITY ERROR */
#define JUNKYS_BAD 0x0049802a /* "you blew it" */
#define JUNKYS_GROSS 0x00498032 /* "this is most unfortunate" */

/* .SEVERITY FATAL */
/* .BASE 64 */
#define JUNKYS_ALLOVER 0x00498204 /* "you bought the farm" */
#define JUNKYS_UNREAL 0x0049820c /* "what is your major?" */
/* .END */
```

If this file is included in a C source file, the C program can refer directly to symbols such as JUNKYS_OK, and if this value is passed as an argument to vos_report_error(), the message printed out will be "everything is ok". Presumably these message symbols are only part of a larger package of routines, and the symbols represent status values that can be returned by these routines to indicate various conditions and results.

The file "test.c" defines tables which are not used directly by the users, and therefore their format is of no interest here. The vos library contains the routines, such as vos_report_error() and vos_translate_error(), which utilize these tables to convert error codes into messages. As part of the user program's initialization, the connection between these vos routines and the tables in "test.c" must be established by calling a routine of the form yyy_register_errors(), where "yyy" is replaced by whatever facility name the user defines in the file "test.msg". (In the sample data shown above, the facility name is JUNKY, so the routine junky_register_errors() would be called as part of the user program's initialization.) The definition of this routine can be found in the file "test.c".

This discussion, the sample output, and Fig. 2 has been in terms of the C language. The only difference for Fortran is that a file "test.for" is generated in addition to "test.h". This file contains the symbolic error codes in the form of Fortran-77 PARAMETER statements suitable for inclusion in a Fortran source program. The tables in "test.c" are independent of the language used by the application program that refers to them, since they are compiled separately.
5. LANGUAGE BINDINGS

The vos routines are designed to be called from any language, but because each language has its own peculiar way of passing parameters and dealing with strings, a "language binding" is defined for each. These bindings have been designed so that they can be used in the programming style that is normal and natural in that language, without reference to other languages. The first version of vos defines bindings for C and Fortran.

The C bindings are macros defined in the file <vos/vos_macros.h>, which must be included in any C source file that references vos routines. The file <vos/vos_def.h> should also be included, since it contains the definitions of all the symbolic constants used by vos.
All strings passed as parameters from C programs to vos functions must be null-terminated, as is the standard C convention, and vos will always null-terminate any strings it passes back as results from a vos function. Furthermore, strings returned by vos functions are either always a fixed length, or there is a second input parameter to that function by means of which the caller can tell vos the maximum number of characters to return in the string. In the latter case, a third parameter to the function will be used by vos to return the actual length of the string. All other parameters passed to vos functions follow the normal C conventions. In particular, input parameters are always passed by value, and output parameters by explicit address reference using the ampersand (&) operator.

The Fortran bindings are subprogram entry points contained in the vos library. The Fortran programmer needs to include the file <vos/vos_def.inc> in his or her Fortran source program in order to obtain the definitions of the vos symbolic constants.

All strings passed as parameters from Fortran programs must be Fortran-77 CHARACTER strings. Fortran always pads out these strings to their full length with trailing blanks, and this convention is followed by vos on all strings it returns from vos functions. The length of a return string is itself returned in another parameter, and this value will not include the trailing blanks used to pad out a string to its full length. All other parameters passed to vos functions follow the normal Fortran conventions.

6. OBJECTS

The principal, unifying feature of vos is its notion of a generalized "object". An object is an abstraction, generalized from the notion of an I/O device, that serves as the focus for most of the vos operations. An object serves as a means by which a process can interact with the external world, including device I/O and network operations, can synchronize its activities, both internally and externally with other processes, and can deal with real time. Table 2 gives the objects that are included in the first version of vos.

<table>
<thead>
<tr>
<th>Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects in VOS</td>
</tr>
<tr>
<td>- Alarm Clock</td>
</tr>
<tr>
<td>- Binary Semaphore</td>
</tr>
<tr>
<td>- Client Input</td>
</tr>
<tr>
<td>- Client Output</td>
</tr>
<tr>
<td>- Event</td>
</tr>
<tr>
<td>- Keyboard</td>
</tr>
<tr>
<td>- Magnetic Tape</td>
</tr>
<tr>
<td>- Notification</td>
</tr>
<tr>
<td>- Process</td>
</tr>
<tr>
<td>- Screen</td>
</tr>
<tr>
<td>- Server Input</td>
</tr>
<tr>
<td>- Server Listening Post</td>
</tr>
<tr>
<td>- Server Output</td>
</tr>
<tr>
<td>- Shared Memory Segment</td>
</tr>
<tr>
<td>- Termination</td>
</tr>
</tbody>
</table>

An object is created by a vos primitive of the form vos_open_xxx(), where "xxx" is replaced by the name of a class of objects, such as a "semaphore", an "event", an interactive "keyboard", a display "screen", etc. For some classes, vos_open_xxx() will map the newly created internal object onto an external device or entity, thereby allowing the
programmer to access the external world via the object abstraction. If successful, the 
\texttt{vos\_open\_xxx()} will create a new instance of an object in the indicated class, and will 
return a "handle" to this object. The handle is simply a small, non-negative number by 
which this object can be referenced in all subsequent vos operations on it. An object is de-
stroyed when a process calls \texttt{vos\_close()} that references the object's handle. This does not 
destroy any external device or entity to which this object was mapped, it just destroys the 
internal object and prohibits further operations on it.

Operations on most objects are initiated by a call of the form \texttt{vos\_start\_zzz()}, where 
"zzz" is replaced by the name of the type of operation desired, such as "read", "write", 
or "listen". In keeping with the requirements for real-time response, all operations in vos 
are inherently asynchronous, which is why the word "start" is included in the primitive. 
The idea is that once an operation is started by \texttt{vos\_start\_zzz()}, the indicated object will 
proceed asynchronously of, and in parallel with, the program issuing the call. There are two 
ways in which a process can detect the completion of an object's operation: it can wait for 
it, or it can associate with the object an asynchronous routine such that when an operation is 
complete, the program will be interrupted and this routine will be activated.

There are two primitives by which a process can wait for an object: \texttt{vos\_wait\_single()}, 
and \texttt{vos\_wait\_multiple()}. The \texttt{vos\_wait\_single()} waits for just a single object to complete, 
the \texttt{vos\_wait\_multiple()} waits for any one of a list of objects to complete. Because an 
object's operation could complete before or after the process issues the wait, each object 
contains an "event flag" that is cleared when an operation is started on an object, and is set 
when the operation completes. The wait primitives simply test the state of this flag, returning 
control to the calling process immediately if the flag is set, and otherwise putting the 
process to sleep until the flag is set by eventual completion of the operation.

The second method for detecting the completion of an object's operation is by the use 
of "ast routines". An ast routine is a subroutine that is associated with an object by the 
\texttt{vos\_attach\_ast()} primitive. Once associated, this routine will be activated whenever an 
operation on the object completes for any reason. This activation is asynchronous because 
it will cause the process to be interrupted, and control to be switched to the ast routine. 
When the ast routine finishes, the interrupted code is continued from the point of interrup-
tion. Only one ast routine at a time can be active for any process, and vos maintains a 
queue of ast routines that are waiting to be activated.

It is possible for a process to utilize both ast routines and waits on the same object at 
the same time. Completion of an object's operation first sets its event flag, then activates its 
ast routine. If the process was waiting for the operation, the wait will be completed when 
the ast routine returns.

Table 2 lists the objects included in vos. We will discuss most of these in the follow-
ing sections, giving examples of their use.

7. I/O DEVICE OBJECTS

I/O device objects in vos are considered "virtual devices" that are mapped onto 
drivers for real devices in the host system at the time the vos object is created. Each device 
is capable of certain operations specific to that device, and only one operation at a time can 
be performed by the device. One operation must be completed before the next operation is 
attempted. The unit of data transmitted by a read or write is a "virtual block". The prop-
erties of this block are determined by the individual device.
7.1. Keyboard Object

An example of an I/O device is the keyboard, which is an input-only object mapped to the keyboard of the user's interactive terminal. A new keyboard object is created and mapped to the interactive keyboard by a C statement of the form:

\[ \text{status} = \text{vos_open_keyboard}(&\text{handle}); \]

where:

\&\text{handle} is the address of an integer into which vos will return a small integer that can be used to reference this keyboard object in all subsequent vos operations.

A read from the keyboard is started by a C statement of the form:

\[ \text{status} = \text{vos_start_read}(\text{handle}, \text{buffer}, \text{max}, \text{param}); \]

where:

\text{handle} is the handle of the keyboard object returned by \text{vos_open_keyboard}().
\text{buffer} is a buffer into which the data will be read.
\text{max} is the maximum number of characters that \text{buffer} will hold.
\text{param} is an arbitrary value that will be passed by vos to any ast routine associated with this object when this operation completes, and that will also be returned from the next wait operation on this object.

A single read operation will transmit into the \text{buffer} one full line of data as typed by the user, including the new-line character at the end of the line (if the user types one). The characters are echoed by the host system as they are typed, and any intra-line editing available in the host system can be performed on this line. (This means that the vos applications automatically adopt the keyboard input conventions of the current host, which are probably not the same on different hosts.) No more than \text{max} characters will be stored in \text{buffer}, and if an input line is longer than \text{max}, only the first \text{max} characters will be delivered to the \text{buffer} – the rest will be kept until the next read operation.

The \text{vos_start_read}() operation initiates the reading, but does not suspend the process until it is complete. Control will return immediately to the calling program, which is then free to proceed in parallel on other calculations. When the program wants to actually utilize the data from the keyboard, it must explicitly wait for the user to type it. This is done by executing a C statement of the form:

\[ \text{status} = \text{vos_wait_single}(\text{handle}, \text{timeout}, &\text{actual}, &\text{param}); \]

where:

\text{handle} is the handle of the keyboard object returned by \text{vos_open_keyboard}().
\text{timeout} is the maximum number of milliseconds to wait for the object to complete its operation.
&\text{actual} is the address of an integer into which vos will store the actual number of characters read into \text{buffer}.
&\text{param} is the address of an integer into which vos will store the value of the \text{param} argument to \text{vos_start_read}().

The value returned by \text{vos_wait_single}() is the host system's status produced by the read operation unless the timeout period elapses before the operation completes, in which
case the status value VOS_TIMEOUT is returned. The value of the timeout parameter can be the symbolic constant VOS_NOTIMEOUT if the process wishes to wait indefinitely for the object to complete its operation. If the value of timeout is zero, there is no wait at all – vos_wait_single() will return immediately with the value VOS_TIMEOUT if the operation is not complete, and with the system’s status value for the operation if it has completed. This allows a program to “poll” the keyboard for input. Note that since keyboard data is being transmitted on a line by line basis, the operation is not considered “complete” until the user has typed a character indicating “transmit line”. This is usually the carriage return character, but, depending on the host, other characters can also have the effect of terminating the read operation. Until the program has explicitly received an indication that the operation has completed (via vos_wait_single() for example), it should not attempt to utilize the data in the buffer since it may not be in a well defined state. Likewise, the data may not be well defined if the status value returned by vos_wait_single() indicates that an error has occurred.

A variation of the vos_start_read() routine is vos_start_read_string(), which is more likely to be used for reading from a keyboard device since it is character oriented. These two routines have exactly the same parameters, and cause exactly the same input operation to occur. The difference is that vos_start_read_string() considers the buffer parameter to be a string, and will store the data into it in the form of a string in the language being used. If this language is C, vos will add a null character after the last data character (which is usually the new-line character). If the language is Fortran, buffer must be a Fortran-77 string descriptor, and vos will pad out any unused characters in the string with trailing blanks.

7.2. Screen Object

The screen object is created and mapped to the user’s interactive terminal display screen with a C statement of the form:

\[
\text{status} = \text{vos}\_\text{open}\_\text{screen}(&\text{handle});
\]

where:

&handle is the address of an integer into which vos will return a small integer that can be used to reference this screen object in all subsequent vos operations.

A write to the screen is started by a C statement of the form:

\[
\text{status} = \text{vos}\_\text{start}\_\text{write}(&\text{handle, buffer, size, param});
\]

where:

handle is the handle of the screen object returned by vos_open_screen().
buffer is a buffer from which the data will be written.
size is the number of characters from buffer to be written.
param is an arbitrary value that will be passed by vos to any ast routine associated with this object when this operation completes, and that will also be returned from the next wait operation on this object.

A single write operation will transmit from the buffer as many lines of data as there are new-line characters in buffer – it neither adds nor removes new-line characters at the end of buffer.

The vos_start_write() operation initiates the writing, then returns control immediately to the calling program, which is free to proceed in parallel on other calculations. During
this time the program must NOT change the data in buffer because it is NOT copied by vos_start_write(). When the program wants to start overwriting the data in buffer, it must explicitly wait for the write operation to complete by calling vos_wait_single().

Related to vos_start_write() is the function vos_start_write_string(), which is called from C by a statement of the form:

\[ \text{status} = \text{vos_start_write_string}(	ext{handle, buffer, param}); \]

where:

- **handle** is the handle of the screen object returned by vos_open_screen().
- **buffer** is a string from which the data will be written. If called from C, this string must be an array of char terminated by a null character. If called from Fortran, buffer must be a Fortran-77 character string descriptor.
- **param** is an arbitrary value that will be passed by vos to any ast routine associated with this object when this operation completes, and that will also be returned from the next wait operation on this object.

Because the length of the string can be determined from the value of buffer, there is no size parameter to this function.

8. I/O EXAMPLES

Below are four sample C programs that use the keyboard object for input and the screen object for output. These programs simply read input lines and echo them out to the screen, which is not very exciting, but it does illustrate all the elements necessary to use vos I/O. The same structures and programming techniques can be used with all I/O devices, including magnetic tapes, network connections, CAMAC and FASTBUS devices, etc.

The first program uses vos_wait_single() to synchronize its I/O objects; the second uses the synchronous vos_read() and vos_write() routines, which are most suitable for this simple demo; the third uses ast routines; and the fourth uses ast routines and multiple buffers.

8.1. I/O using waits

The first program begins by opening both the keyboard and screen objects, getting back handles to them in the integer variables in and out respectively. It then enters a loop where the first operation is to start reading a string from the keyboard into the buffer character array. Note that in C this array must be declared to be one byte bigger than buf_size in order to accommodate the null termination character vos will add to the end of the string it reads (which will contain a maximum of buf_size characters). After starting the read, we immediately call vos_wait_single() to wait indefinitely for the operation to complete. If the user types an end of file indication, the value of status returned by vos_wait_single() will be VOS_ENDOFFILE and we will exit the loop via the break statement. Otherwise, we start writing the string just read to the screen and then wait for that write to complete. The loop repeats until a line containing no characters is read (i.e., an empty line), at which point the loop terminates. We clean up by closing the keyboard and screen objects, then call vos_terminate() to terminate our program.
/* terminal11.c */
/* illustrate vos terminal I/O using waits */
#include <rdr/vos/vos_macros.h>
#include <rdr/vos/vos_def.h>

#define buf_size 256

int main()
{
    int status, inh, outh, size, param;
    char buffer[buf_size+1];

    /* first create keyboard and screen objects */
    vos_report_error(vos_open_keyboard(&inh));
    vos_report_error(vos_open_screen(&outh));

    do /* start reading from keyboard into buffer */
    {
        status = vos_start_read_string(inh, buffer, buf_size, 0);
        if( vos_failure(vos_report_error(status)) )
            break;

        /* wait for read to complete */
        status = vos_wait_single(inh, VOS_NOTIMEOUT, &size, &param);
        if( status == VOS_ENDOFFILE || vos_failure(vos_report_error(status)) )
            break;

        /* write out to screen what was just read */
        status = vos_start_write_string(outh, buffer, 0);
        if( vos_failure(vos_report_error(status)) )
            break;

        /* wait for write to complete */
        status = vos_wait_single(outh, VOS_NOTIMEOUT, &size, &param);
        if( vos_failure(vos_report_error(status)) )
            break;
    } while( size > 0 );

    /* close up the keyboard and screen */
    vos_report_error(vos_close(outh));
    vos_report_error(vos_close(inh));
    vos_terminate(0);
}

8.2. Synchronous I/O

This first version of the program does not really need the asynchronous I/O features of vos, because it waits for the completion of an I/O operation immediately after starting that operation, performing no useful work in between. Because this happens fairly often, vos provides synchronous I/O routines that simply start an I/O operation and then wait for it to complete. For example, the routine vos_read_string() is exactly equivalent to:
#include <rdr/vos/vos_cmacros.h>
#include <rdr/vos/vos_def.h>

int vos_read_string( handle, buf, max, actual )
    int    handle, max, *actual;
    char   *buf;
    { 
        int    status;

        *actual = 0;
        status = vos_start_read_string(handle, buf, max, 0);
        if( vos_failure(status) )
            return status;
        return vos_wait_single(handle, VOS_NOTIMEOUT, actual, &status);
    }

Using the synchronous I/O routines gives the following simplified version of the same
program for keyboard input and screen output.

/* terminal2.c */

/* illustrate vos terminal I/O using synchronous I/O */
#include <rdr/vos/vos_cmacros.h>
#include <rdr/vos/vos_def.h>

#define buf_size 256

int main()
    { 
        int status, inh, outh, size, param;
        char buffer[buf_size+1];

        /* first create keyboard and screen objects */
        vos_report_error(vos_open_keyboard(&inh));
        vos_report_error(vos_open_screen(&outh));

        do (/* read from keyboard into buffer */
            status = vos_read_string(inh, buffer, buf_size, &size);
            if( status == VOS_ENDOFFILE || vos_failure(vos_report_error(status)) )
                break;

            /* write out to screen what was just read */
            status = vos_write_string(outh, buffer);
            if( vos_failure(vos_report_error(status)) )
                break;
        }
        while( size > 0 );

        /* close up the keyboard and screen */
        vos_report_error(vos_close(outh));
        vos_report_error(vos_close(inh));
        vos_terminate(0);
    }
8.3. I/O using ast routines

The third version of this program uses ast routines to process all I/O completely asynchronously. After attaching ast routines to each I/O device and initiating the first read from the keyboard, the main program becomes a "background" task, which could perform any type of independent computation. In this example, the main program simply goes to sleep until the I/O is no longer "busy", at which point it detaches the ast routines, closes the devices and terminates. The code for the main program and the global declarations is:

```c
/* terminal3.c */

/* illustrate vos terminal I/O using asts */

#include <rdr/vos/vos_cmmacros.h>
#include <rdr/vos/vos_def.h>

#define buf_size 256

static int status, inh, outh, busy;
static char buffer[buf_size+1];

#include "routines3.c"

int main()
{
    /* first create keyboard and screen objects */
    vos_report_error(vos_open_keyboard(&inh));
    vos_report_error(vos_open_screen(&outh));

    /* now attach ast routines to these objects */
    vos_attach_ast(inh, keyboard_routine);
    vos_attach_ast(outh, screen_routine);

    /* start reading from keyboard into buffer */
    busy = 1;
    status = vos_start_read_string(inh, buffer, buf_size, 0);
    if( vos_failure(vos_report_error(status)) )
        vos_terminate(0);

    /* process things in the background */
    while( busy )
        vos_sleep(1000);

    /* now detach ast routines to these objects */
    vos_detach_ast(inh);
    vos_detach_ast(outh);

    /* close up the keyboard and screen */
    vos_report_error(vos_close(outh));
    vos_report_error(vos_close(inh));
    vos_terminate(0);
}
```

The ast routine for the keyboard in this example is called `keyboard_routine()`. As with all ast routines, it will be activated whenever an operation on this object completes, at which time it will be passed four parameters:
handle the handle of the object which caused this activation.
status the status code of the operation just completed.
actual the actual number of characters transmitted in the operation just completed.
param the parameter supplied by the user to the vos_start_zzz function that started the operation just completed.

For keyboard_routine(), the value of handle should be the same as the value of the global variable inh, the value of actual will be the number of characters read into buffer, and the value of param will be 0, because this was the value of the param argument to vos_start_read_string() issued in the main program. The keyboard ast routine first checks the value of status. If it is VOS_ENDOFFILE or a failure value, this routine simply sets the global variable busy to zero, indicating that the I/O is finished. Otherwise, it calls vos_start_write_string() to start writing to the screen the buffer that was just read. If this write cannot be started successfully, the busy flag is also set to zero to indicate that I/O has finished. Note that vos_report_error() is called with the value of status to report any errors to the output device. Once the write to the screen has started, this ast routine is finished and it returns (to vos). Control then resumes in the interrupted main task.

When the write to the screen completes, the background task will again be interrupted and the ast routine for the screen, called screen_routine(), will be activated, again being passed the four parameters listed above. Screen_routine() takes analogous actions to keyboard_routine(): it first checks the value of status to see if the operation completed successfully. If it failed, the global busy flag is set to zero. If it succeeded, another read from the keyboard is started into buffer. If this read cannot be started, busy is set to zero, but if the read is started successfully, screen_routine() returns, which will cause vos to resume execution of the interrupted background task.

/* routines3.c */

void keyboard_routine(handle, status, actual, param)
{ int handle, status, actual, param;
  if( status == VOS_ENDOFFILE || vos_failure(vos_report_error(status)) )
    busy = 0;
  else
    { /* write out to screen what was just read */
      status = vos_start_write_string(outh, buffer, 0);
      if( vos_failure(vos_report_error(status)) )
        busy = 0;
      };

void screen_routine(handle, status, actual, param)
{ int handle, status, actual, param;
  if( vos_failure(vos_report_error(status)) )
    busy = 0;
  else
    { /* start reading from keyboard into buffer */
      status = vos_start_read_string(inh, buffer, buf_size, 0);
      if( vos_failure(vos_report_error(status)) )
        busy = 0;
      };
}
8.4. I/O using ast routines and multiple buffers

The third version of the terminal I/O program has a single buffer which is either being filled by a read from the keyboard, or is being emptied by a write to the screen. There is no overlap of the I/O, because the write cannot be started until the read finishes (since otherwise there would be nothing to write), and the next read cannot be started until the write finishes (since otherwise the contents of buffer might get destroyed before they were written to the screen). For a slow device, such as a terminal, this kind of I/O is perfectly acceptable. However, I/O in high energy physics experiments is neither slow, nor does it occur is such strictly alternating input and output sequences. More likely the data is "bursty", meaning that there will be a rush of input data from detectors, followed by a relatively long quiescent period before the next burst of input data. During the input burst, input data rates will be higher than can be sustained on the output device, such as a tape drive, and the data must be stored in internal buffers until the quiet period, when the output device can "catch up" and write it all out.

We will illustrate several techniques for dealing with this type of situation. The first, which is the simplest, is just a fourth version of the terminal I/O program in which there is a set of nbufs buffers. Each buffer is in one of four states:

- **Empty**: the buffer contains no useful data.
- **Filling**: data is actively being read from the keyboard into this buffer.
- **Full**: the buffer contains data that has not yet been written out.
- **Emptying**: data is actively being written to the screen from this buffer.

All buffers are initially in the **empty** state. One of the buffers is put into the **filling** state when a vos_start_read_string() operation is performed on it. When this read operation completes, this buffer is put into the **full** state. It changes from **full** to **emptying** when a vos_start_write_string() operation is performed on it, and when this write operation completes, the buffer again becomes **empty**. The states of a buffer, and the operations that cause a buffer to change state, are illustrated in the following diagram.

![Buffer States and Life Cycle Diagram](image)

**Fig. 3** Buffer states and life cycle

Buffers are moved through their "life cycles" by the actions of the ast routines for the keyboard and screen objects. The main program starts everything off by issuing a
vos_start_read_string() operation on the first buffer. All subsequent I/O operations are performed by the ast routines. When the read completes, the ast routine for the keyboard is activated. Assuming no errors, this routine must do two things:

(1) it must start the write operation necessary to print this buffer to the screen.
(2) it must start a new read operation from the keyboard into the next empty buffer.

The idea is to get both the input (into the next empty buffer) and the output (of the previous full buffer) going simultaneously. There are, however, two problems of coordination that must be solved.

(1) A write cannot be started if the screen object is still busy writing a previous buffer.
(2) A new read cannot be started if there are no empty buffers.

In order to coordinate these operations properly, we introduce four global variables:

nempty   the number of buffers in either the empty or filling state.
nfull    the number of buffers in either the full or emptying state.
inbusy   a flag that indicates the status of the keyboard object.
outbusy  a flag that indicates the status of the screen object.

The status flags, inbusy and outbusy, can take on one of three values:

+1       the device is busy performing an operation.
0        the device is idle.
-1       the device is stopped due to an error.

Initially, both inbusy and outbusy are set to zero, indicating that both objects are idle. Also, the initial value of nfull is set to zero, and the initial value of nempty is set to nbufs, since all buffers are initially empty. In general, the relationship: nfull + nempty = nbufs will always hold.

The code for the main program is very similar to that for the main program of version 3. We initialize the global variables as described above, create the keyboard and screen objects, attach ast routines to each object, start a read into the first buffer, and then do some background processing, which in this simple example is just sleeping until all I/O has stopped (i.e., both inbusy and outbusy become non-positive).

/* terminal4.c */
/* illustrate vos terminal I/O using asts and multiple buffers */

#include <rdr/vos/vos_cmacros.h>
#include <rdr/vos/vos_def.h>

#define buf_size 256
#define nbufs 4

/* inbusy and outbusy can take on 3 values:
 +1 device is busy
 0  device is idle
-1  device stopped due to error
should always have: nfull + nempty = nbufs */

static int status, inh, outh, nfull, nempty, inbusy, outbusy;
static char buffer[nbufs][buf_size+1];
#include "starts4.c"
#include "routines4.c"

int main()
{
    /* first initialize global variables */
    nfull = 0;
    nempty = nbufs;
    inbusy = 0;
    outbusy = 0;

    /* now create keyboard and screen objects */
    vos_report_error(vos_open_keyboard(&inh));
    vos_report_error(vos_open_screen(&outh));

    /* now attach ast routines to these objects */
    vos_attach_ast(inh, keyboard_routine);
    vos_attach_ast(outh, screen_routine);

    /* start reading from keyboard into buffer */
    if(!start_a_read(0))
        vos_terminate(0);

    /* process things in the background */
    while( inbusy > 0 || outbusy > 0 )
        vos_sleep(1000);

    /* now detach ast routines to these objects */
    vos_detach_ast(inh);
    vos_detach_ast(outh);

    /* close up the keyboard and screen */
    vos_report_error(vos_close(outh));
    vos_report_error(vos_close(inh));
    vos_terminate(0);
}

We have declared the set of buffers as the array buffer, which has nbufs elements, each of which can hold buf_size+1 characters. The buffers are indexed by the small integers 0, 1, 2, ..., nbufs–1. The buffer index will be passed as the value of the param argument to the vos_start_read_string() and vos_start_write_string() operations so that when these operations complete, this index will be passed as the value of the param parameter to the ast routine for the object, which is how that routine finds out which buffer was just filled or emptied. Given that an operation just completed on the buffer indexed by param, the next operation of the same type (i.e., the next read operation if a read just completed) should be performed on the buffer indexed by (param + 1)%nbufs, provided that this buffer is in the correct state (i.e., empty to begin a read operation). We do not explicitly represent the state of each buffer as a separate variable, which would be necessary if the buffers were used in random order, because the buffers are used in a strict sequence (0, 1, 2, ..., nbufs–1, 0, 1, 2, ..., nbufs–1, 0, ...). We can therefore infer the state of the next buffer from the value of nfull or nempty as appropriate (i.e., if nempty is positive, then the next buffer to be read in sequence must, in fact, be empty).

Consider first keyboard_routine(), the ast routine for the keyboard object. As in the previous version, this routine starts by checking the value of the parameter status to see if either the end of file or an error was detected, in which case inbusy is set to −1. If the read
operation completed successfully, `keyboard_routine()` increments `nfull` and decrements `nempty` (because it just filled a buffer), and then sets `inbusy` to zero (because the keyboard is now idle). It then attempts to start writing to the screen the buffer that was just filled, and then attempts to start reading from the keyboard into the next empty buffer.

The ast routine for the screen object, `screen_routine()`, has an analogous structure. It first checks the value of `status` for an error, setting `outbusy` to -1 if there was one. If the operation completed successfully, `nfull` is decremented and `nempty` is incremented (because we just emptied a buffer), and then `outbusy` is set to zero (because the screen is now idle). We then attempt to start writing out the next full buffer in sequence, followed by an attempt to start a read into the buffer we just emptied.

Since vos allows only one ast routine at a time to be active, neither `keyboard_routine()` nor `screen_routine()` can be interrupted once it has begun executing. This guarantees that when one of these routines modifies common variables, such as `nempty` and `nfull`, there is no interference from the other routine, and therefore the values of these variables will always be consistent.

```c
/* routines4.c */

void keyboard_routine(handle, status, actual, param)
    int handle, status, actual, param;
    {
        if( status == VOS_ENDOFFILE )
            inbusy = -1;
        else if( vos_failure(status) )
            { vos_report_error(status);
              inbusy = -1;
            }
        else
            { nfull++;
              nempty--;
              inbusy = 0;
              if( start_a_write(param) )
                  start_a_read(((param + 1)%nbufs));
            }
    }

void screen_routine(handle, status, actual, param)
    int handle, status, actual, param;
    {
        if( vos_failure(status) )
            { vos_report_error(status);
              outbusy = -1;
            }
        else
            { nfull--;
              nempty++;
              outbusy = 0;
              if( start_a_write((param + 1)%nbufs) )
                  start_a_read(param);
            }
    }
```
The last two routines, start_a_read() and start_a_write(), encapsulate the bookkeeping necessary to start reading from the keyboard and writing to the screen respectively. Both accept as a parameter the index of the buffer on which the operation is to be performed. Start_a_read() will actually call vos_start_read_string() only if the two coordinating conditions are met: the keyboard object is idle, and the indicated buffer is actually empty. Likewise, start_a_write() will actually call vos_start_write_string() only if the screen object is idle and the indicated buffer is actually full. If the vos_start_read_string() routine returns an error status, start_a_read() reports the error, sets inbusy to -1 to mark the object as "stopped due to an error", and returns a value of false (zero). If the vos_start_read_string() is successful, inbusy is set to one to mark the object as busy, and start_a_read() returns a value of true (one). The actions of start_a_write() are analogous.

/* starts4.c */

/* Try to start a read on buffer indexed by which. */
/* Returns true (1) if ok, false (0) if error. */
int start_a_read(which)
    int which;
    {
        int status;

        if( inbusy == 0 && nempty > 0 )
            {
                status = vos_start_read_string(inh,buffer[which],buf_size,which);
                if( vos_failure(status) )
                    {
                        vos_report_error(status);
                        inbusy = -1;
                        return 0;
                    }
                inbusy = 1;
            }
        return 1;
    }

/* Try to start a write on buffer indexed by which. */
/* Returns true (1) if ok, false (0) if error. */
int start_a_write(which)
    int which;
    {
        int status;

        if( outbusy == 0 && nfull > 0 )
            {
                status = vos_start_write_string(outh,buffer[which],which);
                if( vos_failure(status) )
                    {
                        vos_report_error(status);
                        outbusy = -1;
                        return 0;
                    }
                outbusy = 1;
            }
        return 1;
    }
9. PROCESSES

One of the most useful concepts in real-time systems is the "process". Each process is a separate program in execution with its own data and code. Although processes are generally independent from each other, there are means by which one process can control another, can communicate with another, and can share common memory. In vos, one process, called the "parent", can create another process, called the "child", with the vos_open_process() primitive. Included as parameters to this primitive are the name of the program which the new child process is to execute, its execution priority, and a list of parameters that are passed to that program when it starts execution. This primitive returns to the parent the process identification assigned to the child by the host system, and a handle for a vos process object.

Once created by a vos_open_process(), the newly created child process is essentially independent of the parent, and executes "in parallel" with it subject to the scheduling algorithm of the host system. A parent process has only limited control over a child process it has created:

1. it can learn when the child process terminates by either waiting on the process object, or attaching an ast routine to this object which will be activated when the child terminates.

2. it can force the child to terminate by calling the vos_terminate() primitive with the child's process identification.

3. it can obtain the child's execution priority by calling vos_get_priority().

4. it can change the child's execution priority by calling vos_set_priority(), although this is subject to certain restrictions on some host systems.

All other forms of interaction between a parent and child are accomplished by more general vos primitives which are discussed in the next sections. These are followed by a section devoted to an example of spawned processes that communicate via shared memory and semaphores.

In vos, priorities are referenced by symbolic names, such as VOS_PRIORITY_LOW and VOS_PRIORITY_HIGH, in order to keep these values portable between hosts that use different numeric values for assigning priorities.

10. SEMAPHORES

Vos provides binary semaphores as the mechanism for locking and unlocking access to a shared resource. A semaphore is a named structure known to all processes running on the host system. It "lives" independently of all processes, although it is first created by a process and can be removed at any time by a process. A binary semaphore is always in one of two states: "locked" and "unlocked". There are only two operations on a semaphore: one to lock it, and one to unlock it. If a process attempts to lock a semaphore that is already locked, that process is blocked until some other process unlocks this semaphore.

The primary use of semaphores is to control access to a shared resource. If all processes lock the same semaphore before starting to access the shared resource, and unlock that semaphore when they are finished accessing the resource, then it will be guaranteed that only one process at a time will have access to the resource. The section of code accessing the shared resource is called a "critical section", and the guarantee of having at most one process in its critical section at any time is called "mutual exclusion".
Each process wishing to use a semaphore must create an internal semaphore object and map it onto a named semaphore in the underlying host system using a statement in C of the form:

```c
status = vos_open_semaphore(&handle, name, &wascreated);
```

where:

- **&handle** is the address of an integer into which this function will store a small number that can be used to reference this semaphore object in subsequent operations.
- **name** is a string which is the name of the semaphore in the host system.
- **&wascreated** is the address of an integer into which this function will store a 1 if it creates a new semaphore in the host system, and 0 if this semaphore already exists in the host system.

At the time a semaphore is first created by the host system, as indicated by a value of 1 returned in `wascreated`, it is in the unlocked state. If the value returned in `wascreated` is zero, the semaphore may be in either state, depending on what other processes are doing, or have done, to it.

Once created, a semaphore is locked by a C statement of the form:

```c
status = vos_lock(handle);
```

and is unlocked by a C statement of the form:

```c
status = vos_unlock(handle);
```

A semaphore is closed by a C statement of the form:

```c
status = vos_close(handle);
```

Closing a semaphore destroys the vos semaphore object, but does not remove the underlying host system semaphore, because other processes may be using this too. The only way to remove the host system’s semaphore is to use a C statement of the form:

```c
status = vos_remove_semaphore(name);
```

where **name** is the name of the host system semaphore entity (i.e., the same name as used in the call to `vos_open_semaphore()`). If other processes have a semaphore open at the time it is removed, they will receive error status returns from all subsequent operations on it.

### 11. SHARED MEMORY

One of the most common resources to be shared by processes is memory. The basic unit of shared memory is called a “segment”, and can be any size, subject to limitations imposed by the host system. Like a semaphore, a segment is a named entity that exists in the underlying host system. Each process wishing to gain access to a shared segment, must first open the segment by name, and then map it into the process’s virtual address space. From then on, storage in the segment can be addressed relative to a pointer to the base of the segment. Segments are “permanent” structures in the underlying host system in that, once created, they live until they are explicitly removed, even if no processes have them open or are mapped to them.
Each process must create a vos segment object and associate it with a shared memory segment in the underlying host system by issuing a C statement of the form:

```c
status = vos_open_segment(&handle, name, size, oktocreate, &wascreated);
```

where:

- **&handle** is the address of an integer into which this function will store a small number that can be used to reference this segment object in subsequent operations.
- **name** is a string which is the name of the segment in the host system.
- **size** is the number of bytes in the segment. If the host segment is created by this call, it will be created with **size** bytes. If the host segment already exists, it must not be smaller that **size** bytes.
- **oktocreate** should be 1 if the caller wishes to create a host segment of this name if it does not already exist, and 0 otherwise. Since a newly created memory segment contains undefined values (i.e., it is not necessarily all zeros), the process that creates it must be prepared to initialize it.
- **&wascreated** is the address of an integer into which this function will store a 1 if it creates a new segment in the host system, and 0 if this segment already exists in the host system. Presumably, if this return value is 1, the process will immediately initialize the new shared memory.

Once we have obtained a handle to a segment, a process can map that segment, or parts of it, into the process’s virtual address space with a C statement of the form:

```c
status = vos_map(handle, offset, &base, size, access);
```

where:

- **handle** is the handle returned by **vos_open_segment()**.
- **offset** is the number of the first byte in the segment that is to be mapped into the virtual address space.
- **&base** is the address of a pointer variable. If this pointer contains a NULL value, vos will map the segment onto an unused virtual address in the process’s virtual address space and will return this virtual address in **base**. If this pointer is not NULL, it is interpreted as a virtual address onto which vos will attempt to map the segment, subject to whatever restrictions the host system may impose on such mappings.
- **size** is the number of bytes to be mapped. The value of **size** plus the value of **offset** must not be more than the actual size of the shared memory segment.
- **access** has the value VOS_READONLY if the process only wishes to have read access to the segment, and VOS_READWRITE if it wishes both read and write access.

Once mapped into a process’s virtual address space, items in the shared memory segment can be accessed as offsets from the pointer stored in **base**. For example, if we wish to treat a shared segment as an array of integers, we would declare **base** as:

```c
int *base;
```

Then, after calling **vos_map()** with **&base** as an argument, we could index through the array as:
for (i = 0; i < N; i++)
    base[i] = 0;

where we have assumed that there are N integers in the shared segment (i.e., the value of the size argument to vos_map() was N \times \text{sizeof(int)}).

Using shared segments from Fortran is a bit more challenging, since Fortran does not have the concept of a pointer. The following code illustrates how a Fortran array can be used to reference a shared segment, provided Fortran subscripts are not checked for being within the array bounds at run time. We must declare an array \( A \) in a common block to get it on a page boundary:

\[
\text{COMMON /XYZ/ } A(10000)
\]

Then we map the shared segment into an unused virtual address:

\[
\text{BASE} = 0
\]

\[
\text{STATUS}=\text{VOS}\_\text{MAP}(\text{HANDLE}, 0, \text{BASE}, 10000, \text{VOS}\_\text{READWRITE})
\]

Then we calculate an adjustment that will be used for all references to array \( A \):

\[
\text{ADJUST} = \text{BASE} - \text{VOS}\_\text{LOC}(A)
\]

Now whenever we wish to reference the \( i^{th} \) element of array \( A \), we use:

\[
A(\text{ADJUST}+i)
\]

A segment is unmapped by a C statement of the form:

\[
\text{status} = \text{vos_unmap(handle)};
\]

and is closed by:

\[
\text{status} = \text{vos_close(handle)};
\]

The vos_close() does not remove the shared memory segment from the underlying host operating system, it merely ends the access which this process has to it. To remove the host system's segment requires a statement of the form:

\[
\text{status} = \text{vos_remove_segment(name)};
\]

where \text{name} is the name of this segment in the host system (i.e., the same name as used in a vos_open_segment() call). Removal of the host’s segment is delayed until all processes accessing that segment have either terminated or closed the segment.

12. PROCESS EXAMPLES

In the fourth version of the I/O example we illustrated how a single process could use ast routines and multiple buffers to gain maximum overlap of input and output operations while being free to perform independent "background" computations. In this section we present a generalization of this problem that is solved in terms of processes, semaphores, and shared memory segments.

In the more general situation, we define a "producer" as a process that generates data. In a real-time application, this data may be read out of detectors through CAMAC or FASTBUS. The important point is that structurally we wish to isolate the details of producing this data into a separate process that operates independently of the rest of the system. Such isolation makes it easier to modify just this part of the software, and makes it easier to
recombine and reuse this software as part of other systems.

The producer process must do something with this data once it has generated it. For this example, we will have it generate the data into a buffer in a shared memory segment so that it can be used by other processes in the system. Since we expect data generation to be bursty, the shared memory segment will contain a "pool" of buffers, each big enough to hold one "event" produced by the producer. To further increase efficiency, we will design the buffering scheme so that the producer will get an empty buffer before producing the next event, thereby enabling the data for the event to be generated directly into the buffer, rather than being generated elsewhere and then being copied into the buffer.

A process that utilizes the buffered data is called a "consumer". In a real-time application, the consumer usually records data onto a permanent storage medium, such as magnetic tape. Again, isolating into a separate process the details of how data are recorded (i.e., the format, labels, identification, etc.) makes the system easier to maintain and modify.

The consumer must pick up the data from every event in exactly the same order as the producer produces it. It is also useful to define a "sampling consumer" as a process which gets data from "as many event buffers as there is time for", without interfering with either the producer or consumer processes. The main data flow is from producer to buffer pool to consumer, but if there is any time left over, the sampling consumer will get copies of as much event data as time allows. Figure 4 illustrates the organization of the three types of process, the shared memory segment, and the data flows between them.

![Diagram of Producer - Consumer problem](image)

Fig. 4 Producer - Consumer problem

Of course, it is possible to have more than one of each type of process, for example one consumer to record data on tape, another to send it out over a network, a third to perform simple analysis, etc. The bookkeeping would be slightly more complex than that shown here for the simple case, but the principles are identical. In an on-line environment we might want to have different scheduling priorities for each type of process: the producer
needs to have highest priority, since it must respond to stimuli in the external world that cannot wait (i.e., it has real-time deadlines); the consumer needs to have the next highest priority, since it must get the data onto tape as fast as possible so that the buffer pool does not fill up and block the producer; and finally, the sampling consumer has lowest priority, since it should get only those CPU cycles that are "left over" by the producer and consumer processes.

We should note that the producer-consumer model is also useful in the off-line environment, and the vos solution developed here (with or without the scheduling priorities) is directly applicable there too. In the off-line situation, the producer process would read events from a tape and would perform a preliminary analysis on them, writing only the "interesting" events into the shared buffer pool. Consumer processes could build up histograms according to various criteria. The benefit of this organization would be the isolation of the analysis from the display processes, making it possible to write one general analysis process that could run with any number of different display processes.

Our solution consists of a main "controller" process and the three data acquisition processes mentioned above, which we call "producer", "consumer", and "sampler" respectively. In our simple example, the controller process will print a "sign-on" message, will create and initialize the shared buffer pool, will create the three data acquisition processes as children, and will wait for them to finish, at which point it will remove the shared buffer pool, and will then terminate itself after printing a "sign-off" message. In a data acquisition system used in an actual experiment, the controller process would obviously be more sophisticated, since it would have to control the starting and stopping of runs, the setting up and recording of run parameters, the interaction with the experimenter, etc.

The C code for the main controller program is as follows:

```c
/* controller.c  *

/* control program for producer consumer problem  */

#include "header.h"
#include "setup_pool.c"
#include "endup_pool.c"

#define prodname "producer"
#define consname "consumer"
#define sampname "sampler"

int main()
{
    int myid, status, segh, semah, actual, param;
    int prodh, prodiid, consh, consid, samph, sampid;
    struct pool *base;

    /* first print sign-on message with my process id */
    vos_get_process_id(&myid);
    printf("controller starting, process id = %d\n", myid);

    /* create and initialize the shared buffer pool */
    setup_pool(&segh, &semah, 1, &base);

    /* spawn the producer, consumer, and sampling consumer processes */
    status = vos_open_process(&prodh, prodname, 0, 0,
                               VOS_PRIORITY_HIGHEST, &prodiid);
```
if( vos_failure(status) )
{
    vos_report_error(status);
    vosTerminate(0);
}

printf(" producer spawned, process id = %d\n", proid);

status = vos_open_process(&consh, consname, 0, 0,
                            VOS_PRIORITY_HIGH, &consid);
if( vos_failure(status) )
{
    vos_report_error(status);
    vosTerminate(0);
}

printf(" consumer spawned, process id = %d\n", consid);

status = vos_open_process(&samph, sampname, 0, 0,
                            VOS_PRIORITY_NORMAL, &sampid);
if( vos_failure(status) )
{
    vos_report_error(status);
    vosTerminate(0);
}

printf(" sampler spawned, process id = %d\n", sampid);

/* wait for the kids to finish */
vos_wait_single(prodh, VOS_NOTIMEOUT, &actual, &param);
vos_wait_single(consh, VOS_NOTIMEOUT, &actual, &param);
base->sampling_ok = 0;
vos_wait_single(samph, VOS_NOTIMEOUT, &actual, &param);

/* close up the shared buffer pool and remove it */
endup_pool(segh, semah, 1);

/* print sign-off message, then terminate */
printf("controller finished, process id = %d\n", myid);
vosTerminate(0);
}

The file header.h, which is included in the source file for all four processes, contains the definition of the structure of the shared memory segment, as well as some symbolic constants defining the size of a buffer, the number of buffers, etc. The shared segment is defined as the C structure pool which contains the following fields:

- nfull: the number of full buffers.
- nempty: the number of empty buffers.
- nsample: the number of buffers available for sampling.
- produce_here: the index to the buffer into which the producer should generate data.
- consume_here: the index to the buffer out of which the consumer should extract data.
- sample_here: the index to the buffer which the sampler should copy for sampling.
- producer_is_waiting: a flag which is non-zero if the producer process is waiting for a buffer to become empty, in which case the consumer must send
a signal to the object called *prodsigname* when it empties the next buffer.

**consumer_is_waiting** a flag which is non-zero if the consumer process is waiting for a buffer to become full, in which case the producer must send a signal to the object called *conssigname* when it fills the next buffer.

**sampling_ok** a flag which is non-zero if the sampler process is allowed to try to sample data.

**signal_connects** a counter of the number of processes set up for signaling.

**buffer** an array of *nbufs* buffers, each holding *buf_size* characters.

The size of this structure is the size of the shared segment, as given by the value of *segsizename*. The value of *segsizename* is the name given to the shared memory segment in the host system. In order to guarantee that processes have exclusive access to this shared segment, we also need a semaphore, the name of which is given by *semaname*.

```c
/* header.h included in all producer/consumer processes */

#include <rdr/vos/vos_cmacros.h>
#include <rdr/vos/vos_def.h>

#define segname "buffer_pool"
#define semaname "buffer_lock"
#define buf_size 256
#define nbufs 4
#define segsize sizeof(struct pool)

/* should always have: nfull + nempty = nbufs */

struct pool {
    int nfull, nempty, nsample;
    int produce_here, consume_here, sample_here;
    int producer_is_waiting, consumer_is_waiting;
    int sampling_ok, signal_connects;
    char buffer[nbufs][buf_size];
};

#define prodsigname "prodsignal"
#define conssigname "conssigname"
```

The producer process begins with a simple "sign-on" message, then maps to the shared buffer pool and sets up the signaling mechanism between itself and the consumer before entering its main loop. This loop consists of three basic steps:

1. **Call get_empty_buffer()** to get the address of an empty buffer into which the producer can generate the data from the next event.
2. **Call fill_it()** to fill the buffer with event data in a manner unique to each experiment.
3. **Call put_full_buffer()** to mark the buffer full so that the consumer and sampler may now process it.

In a real data acquisition system, there would be more elaborate mechanisms for starting, pausing, and stopping a run, but for purposes of this demo we simply allow fill_it() to return a value of zero to indicate that it has filled the buffer for the last event and that the loop should therefore terminate. When control exits the loop, the producer calls
endup_signals() to do whatever is necessary to stop the signals between itself and the consumer process, then calls endup_pool() to unmap itself from the shared buffer pool. Finally, it prints a brief "sign-off" message and terminates itself.

```c
/* producer.c */

/* producer program for producer consumer problem */

#include "header.h"
#include "setup_pool.c"
#include "endup_pool.c"
#include "setup_signals.c"
#include "endup_signals.c"
#include "get_empty_buffer.c"
#include "fill_it.c"
#include "put_full_buffer.c"

int main()
{
    int myid, segh, semah, n, count;
    int prodh, consh;
    struct pool *base;
    char *bufptr;

    /* first print sign-on message with my process id */
    vos_get_process_id(&myid);
    printf(" producer starting, process id = %d\n", myid);

    /* set up and map shared buffer pool */
    setup_pool(&segh, &semah, 0, &base);

    /* set up the signals between producer and consumer */
    setup_signals(base, semah, &prodh, &consh, 1);

    /* main producer loop */
    count = 0;
    do {
        bufptr = get_empty_buffer(base, semah, prodh);
        n = fill_it(bufptr);
        put_full_buffer(base, semah, consh);
        count++;
    } while( n != 0 );

    /* end up the signals between producer and consumer */
    endup_signals(base, semah, prodh, consh);

    /* end up the shared buffer pool */
    endup_pool(segh, semah, 0);

    /* print sign-off message, then terminate */
    printf(" producer finished, %d buffers produced\n", count);
    vos_terminate(0);
}
```

The consumer process is the analog of the producer: it also begins with a simple "sign-on" message, maps itself to the shared buffer pool and sets up the signaling mechanism between itself and the producer before entering its main loop. The consumer’s loop...
also consists of three basic steps:

1. Call `get_full_buffer()` to get the address of the next buffer full of event data to be recorded to tape (or whatever else "consume" might require).

2. Call `empty_it()` to consume the event data in a manner unique to each experiment.

3. Call `put_empty_buffer()` to mark the buffer empty so that the producer can refill it.

In a real data acquisition system, there would be more elaborate mechanisms for starting, pausing, and stopping a run, but for purposes of this demo we simply allow `empty_it()` to return a value of zero to indicate that it has emptied the buffer for the last event and the loop should now terminate. When control exits the loop, the consumer calls `endup_signals()` to do whatever is necessary to stop the signals between itself and the producer process, then calls `endup_pool()` to unmap itself from the shared buffer pool. Finally, it prints a brief "sign-off" message and terminates itself.

```c
/* consumer.c */

/* include program for producer consumer problem */

#include "header.h"
#include "setup_pool.c"
#include "endup_pool.c"
#include "setup_signals.c"
#include "endup_signals.c"
#include "get_full_buffer.c"
#include "empty_it.c"
#include "put_empty_buffer.c"

int main()
{
    int  myid, segh, semah, n, count;
    int  prodh, consh;
    struct pool  *base;
    char    *bufptr;

    /* first print sign-on message with my process id */
    vos_get_process_id(&myid);
    printf(" consumer starting, process id = %d\n", myid);

    /* set up and map shared buffer pool */
    setup_pool(&segh, &semah, 0, &base);

    /* set up the signals between producer and consumer */
    setup_signals(base, semah, &prodh, &consh, 0);

    /* main consumer loop */
    count = 0;
    do 
    { 
        bufptr = get_full_buffer(base, semah, consh);
        n = empty_it(bufptr);
        put_empty_buffer(base, semah, prodh);
        count++;
    } 
    while( n != 0);

    /* end up the signals between producer and consumer */
    endup_signals(base, semah, prodh, consh);
```
/* end up the shared buffer pool */
endup_pool(segh, semah, 0);

/* print sign-off message, then terminate */
printf(" consumer finished, %d buffers consumed\n", count);
vos_terminate(0);
}

The sampling consumer process, which we call “sampler”, is simpler than the “full” consumer. It also begins with a simple “sign-on” message, and maps itself to the shared buffer pool, but it does not have to set up any signaling mechanism between itself and the producer or consumer because it runs on a “time available” basis that requires no signaling interaction with other processes. The sampler’s loop is also simpler, consisting of only two steps:

1) Call get_sample_buffer() to get the next buffer full of unsampled event data.
2) Call sample_it() to sample the event data in a manner unique to each experiment.

In a real data acquisition system, there would be more elaborate mechanisms for starting, pausing, and stopping a run, but for purposes of this demo we simply define a flag variable in the shared memory segment called sampling_ok that is set to 1 during the initialization of the buffer pool and is set to 0 by the controller after it detects the termination of both the producer and the consumer processes. Control will exit the sampler loop when the value of this flag becomes zero, at which time the sampler calls endup_pool() to unmap itself from the shared buffer pool, then prints a brief “sign-off” message and terminates.

/* sampler.c     */
/* sampling consumer program for producer consumer problem */

#include "header.h"
#include "setup_pool.c"
#include "endup_pool.c"
#include "get_sample_buffer.c"
#include "sample_it.c"

int main()
{
    int myid, segh, semah, n, count;
    struct pool *base;
    char copy[buf_size];

    /* first print sign-on message with my process id */
    vos_get_process_id(&myid);
    printf(" sampler starting, process id = %d\n", myid);

    /* set up and map shared buffer pool */
    setup_pool(&segh, &semah, 0, &base);

    /* main sampling consumer loop */
    count = 0;
    while( base->sampling_ok )
    {
        if( get_sample_buffer(base, semah, copy) )
        {
            sample_it(copy);
            count++;
        }
    }
/* end up the shared buffer pool */
endup_pool(segh, semah, 0);

/* print sign-off message, then terminate */
printf(" sampler finished, %d buffers sampled\n", count);
vos_terminate(0);
}

We have encapsulated all the code necessary to set up the shared memory segment and its controlling semaphore in the routine setup_pool(). This routine is called once by each process wishing to access the pool as part of that process's initialization. The routine has four parameters:

- **&segh** the address of an integer into which this routine will store the handle for the shared segment.
- **&semah** the address of an integer into which this routine will store the handle for the mutual exclusion semaphore.
- **oktocrate** a flag which is 1 if it is okay to create and initialize a new shared memory segment, 0 if not. Only the controller process will pass a value of 1 to this parameter; all other processes will pass a 0 because the initialization should only be done once.
- **&base** the address of a pointer variable into which this routine will store the virtual address of the mapped shared memory structure for this process.

**Setup_pool()** calls vos_open_segment() to create the segment object and associate it with the host's shared memory segment whose name is given by segname. It then calls vos_map() to map this segment into the virtual address space of the process calling setup_pool(). If the segment was just created, which should only happen when oktocrate is 1, all the variables in the pool structure are initialized. Finally, vos_open_semaphore() is called to create the semaphore object that will be used to guarantee mutually exclusive access to the shared segment. This semaphore is initially in the "unlocked" state. Note that when this routine returns control to its caller, the process is left mapped to the shared segment so that it can reference the data stored there via the pointer returned in base.

```c
void setup_pool(segh, semah, oktocrate, base)
{
    int *segh, *semah, oktocrate;
    struct pool **base;
    
    int status, wascreated;

    /* open shared memory segment */
    status = vos_open_segment(segh, segname, segsize, oktocrate, &wascreated);
    if( vos_failure(status) )
    {
        vos_report_error(status);
        vos_terminate(0);
    }

    /* map to the shared segment so we can access the buffer pool */
    *base = (struct pool *)0;
    status = vos_map(*segh, 0, base, segsize, VOS_READWRITE);
    if( vos_failure(status) )
    {
        vos_report_error(status);
        vos_close(*segh);
    }
```
vos_remove_segment(segname);
vos_terminate(0);
}

if( wascreated )
    /* shared segment was just created, initialize it */
    (*base)->nfull = 0;
    (*base)->nempty = nbufs;
    (*base)->nsample = 0;
    (*base)->produce_here = 0;
    (*base)->consume_here = 0;
    (*base)->sample_here = 0;
    (*base)->producer_is_waiting = 0;
    (*base)->consumer_is_waiting = 0;
    (*base)->sampling_ok = 1;
    (*base)->signal_connects = 0;
}

/* also open the semaphore to control access to shared segment */
status = vos_open_semaphore(semah, segname, &wascreated);
if( vos_failure(status) || (wascreated && !oktoremove) )
    /* open failed, or it created it when we did not want to */
    vos_report_error(status);
    vos_unmap(*segh);
    vos_close(*segh);
    vos_remove_segment(segname);
    vos_terminate(0);
}

The complement to setup_pool() is endup_pool(), which should be called once at the end of each process that has called setup_pool() in order to close the shared segment and mutual exclusion semaphore objects. This routine has three parameters:

segh       the handle for the shared segment.
semah      the handle for the mutual exclusion semaphore.
oktoremove a flag which is 1 if it is okay to remove the shared memory segment and its mutual exclusion semaphore, 0 if not. Only the controller process will pass a value of 1 to this parameter; all other processes will pass a 0 because the removal should only be attempted once.

void endup_pool(segh, semah, oktoremove)
{
    int     segh, semah, oktoremove;
    (/* close the semaphore and then remove it if asked */
    vos_report_error(vos_close(semah));
    if( oktoremove )
        vos_remove_semaphore(segname);

    /* unmap and close the shared memory segment, then remove it if asked */
    vos_report_error(vos_unmap(segh));
    vos_report_error(vos_close(segh));
    if( oktoremove )
        vos_remove_segment(segname);
    )
Access to the bookkeeping variables in the shared buffer pool is limited to five routines, each of which constitutes a critical section of code for the process invoking it and therefore each of which must lock the mutual exclusion semaphore on entry and unlock it on exit. The first of these routines is `get_empty_buffer()`, which is called by the producer in order to obtain the address of an empty buffer into which it can generate data for the next event. There are three parameters:

- **base** the pointer to the shared memory segment.
- **buffer_lock** the handle of the mutual exclusion semaphore.
- **wait_object** the handle of the object by means of which the consumer will send a signal to the producer if the producer cannot find any empty buffers.

After locking `buffer_lock` to get exclusive access to the shared memory segment, `get_empty_buffer()` tests the value of `nempty`. If this is positive, that number of buffers are empty at this time. However, if this value is zero (it should never be negative), the consumer is too slow for this producer and the producer must wait for the consumer to free up a buffer. It sets the `producer_is_waiting` flag, so that the consumer knows that a signal must be sent when the next buffer is freed (it is, in general, too expensive to have the consumer simply send a signal whenever it frees a buffer, whether or not the producer is waiting), and then calls `wait_for_signal()` to do the actual waiting. As we will see later, `wait_for_signal()` must unlock the mutual exclusion semaphore just before actually blocking this process, and must lock it again immediately after the process becomes unblocked, so that while it is asleep, the producer does not prevent the consumer from accessing the shared memory. (Obviously, if the producer did prevent the consumer by keeping the shared memory segment locked, the consumer could never access the shared memory to find out that the producer was sleeping and hence could never send the signal to awaken it – the system would be "deadlocked".)

Once the value of `nempty` indicates that a buffer is free, we increment the value of `produce_here`, which is the index to the next buffer in sequence to be filled. We then compute into the local variable where the address of this buffer, and decrement `nempty` by one. Because we might be about to refill a buffer that has not yet been processed by sampler, we next test the value of `nsample`. If this is less than `nbufs`, then this buffer has already been processed by sampler and there is nothing more to do. However, if `nsample` equals `nbufs` (it should never be greater than `nbufs`), then sampler is so slow that the producer has "wrapped around" behind it, filling all the buffers before any have been sampled. Therefore, we must "push" ahead the value of `sample_here`, which is the index of the next buffer to sample, and reduce `nsample` by one because the sampler should not attempt to process the buffer that the producer is currently filling (obviously if this were to happen, the sampler would get inconsistent data from two different events).

Just before leaving, `get_empty_buffer()` unlocks the mutual exclusion semaphore `buffer_lock`. It returns the value of `where`, which is the address of the next empty buffer the producer is to use.
#include "wait_for_signal.c"

char *get_empty_buffer(base, buffer_lock, wait_object)
struct pool *base;
int buffer_lock, wait_object;
{
    char *where;

    /* get exclusive access to shared segment */
    vos_lock(buffer_lock);

    /* check for empty buffer */
    while( base->nempty <= 0 )
    { /* no empty buffers left, wait for one */
        base->producer_is_waiting = 1;
        wait_for_signal(buffer_lock, wait_object);
    }

    /* at least one empty buffer left, use it */
    base->produce_here = (base->produce_here + 1) % nbufs;
    where = base->buffer[base->produce_here];
    base->nempty--;

    /* if necessary, adjust number of buffers left for sampling */
    if( base->nsample >= nbufs )
    {
        base->sample_here = (base->sample_here + 1) % nbufs;
        base->nsample--;
    }

    /* release exclusive access to shared segment */
    vos_unlock(buffer_lock);

    return where;
}

After the producer has filled a buffer, as done by fill_it() in some experiment-specified manner, it calls the routine put_full_buffer() to do the bookkeeping necessary to mark a buffer "full". There are also three parameters to this routine:

**base** the pointer to the shared memory segment.

**buffer_lock** the handle of the mutual exclusion semaphore.

**signal_object** the handle of the object by means of which the producer will send a signal to the consumer if the consumer is waiting for a buffer to become full.

As expected, the first action of this routine must be to lock the mutual exclusion semaphore **buffer_lock**, and its last action is to unlock this same semaphore. In between, it has exclusive access to the shared pool structure. We increment nfull to indicate that another buffer is now full, and nsample to indicate that another buffer is available for sampling. We also test consumer_is_waiting, which will be non-zero if the consumer process has gone to sleep because it could find no full buffers. If we detect this condition, we reset consumer_is_waiting to zero, then send a signal to wake up the consumer process.
void put_full_buffer(base, buffer_lock, signal_object)
    struct pool  *base;
    int        buffer_lock, signal_object;
{
    /* get exclusive access to buffer pool */
    vos_lock(buffer_lock);

    /* mark next buffer "full", available for consuming and sampling */
    base->nfull++;
    base->nsample++;
    if( base->consumer_is_waiting )
        { /* send signal to consumer who is waiting for full buffer */
          base->consumer_is_waiting = 0;
          send_signal(signal_object);
        }

    /* release exclusive access to buffer pool */
    vos_unlock(buffer_lock);
}

The routine put_full_buffer() is similar to get_empty_buffer(), with "full" substituted for "empty", and "consumer" substituted for "producer". The only asymmetry is that put_full_buffer() does not do anything with nsample, since consuming a buffer is totally independent of sampling that buffer. (The code dealing with nsample could be omitted from get_empty_buffer() if there were no sampling consumer processes.)

The routine put_empty_buffer() is analogous to put_full_buffer(): we substitute "empty" for "full", "producer" for "consumer", and remove all references to nsample.

#include "wait_for_signal.c"

char *get_full_buffer(base, buffer_lock, wait_object)
    struct pool  *base;
    int        buffer_lock, wait_object;
{
    char      *where;

    /* get exclusive access to shared segment */
    vos_lock(buffer_lock);

    /* check for full buffer */
    while( base->nfull <= 0 )
        { /* no empty buffers left, wait for one */
          base->consumer_is_waiting = 1;
          wait_for_signal(buffer_lock, wait_object);
        }

    /* at least one full buffer left, use it */
    base->consume_here = (base->consume_here + 1)%nbufs;
    where = base->buffer[base->consume_here];
    base->nfull--;

    /* release exclusive access to shared segment */
    vos_unlock(buffer_lock);
    return where;
}
#include "send_signal.c"

void put_empty_buffer(base, buffer_lock, signal_object)
struct pool *base;
int buffer_lock, signal_object;
{
  /* get exclusive access to buffer pool */
  vos_lock(buffer_lock);

  /* mark next buffer "empty", available for refilling */
  base->nextempty++;
  if( base->producer_is_waiting )
    /* send signal to producer who is waiting for empty buffer */
    base->producer_is_waiting = 0;
    send_signal(signal_object);
  }

  /* release exclusive access to buffer pool */
  vos_unlock(buffer_lock);
}

The fifth and final routine to perform bookkeeping functions on the shared buffer pool is get_sample_buffer(). This is called by the sampler to find out if a buffer is available for sampling and if so, to return a copy of it. This function will return true (1) if a buffer has been found to sample, and false (0) otherwise. There are three parameters:

  base the pointer to the shared memory segment.
  buffer_lock the handle of the mutual exclusion semaphore.
  copy the address of an array into which this routine will copy the next buffer it finds that is available for sampling.

For reasons of efficiency, this routine "cheats" by first testing nsample directly, without gaining exclusive access to the shared memory segment. If this value is not positive, no buffers are available for sampling and the routine returns a zero value to indicate this. This case does not incur the overhead of locking and unlocking the mutual exclusion semaphore. However, if the value of nsample is positive, that many buffers are available for sampling and we then lock the mutual exclusion semaphore buffer_lock before doing anything else. Once we have exclusive access to the shared buffer pool, we again check the value of nsample. In this simple demo, this recheck is not strictly necessary, since the value of nsample cannot be reduced to zero or below by any other process. In general, however, this value could have changed between the time of the test outside the critical section and the time this process gained exclusive access to the critical section, so a recheck is advisable. Should the value of nsample still be positive, we increment sample_here, the index to the next buffer to sample, copy that buffer into the parameter copy, and then reduce the number of buffers remaining to by sampled by one. Although copying the data out of the buffer takes time, and is not done by either the producer or the consumer, it is necessary for the sampler to do it so that the producer is free to reuse the buffer without having to wait for the sampler to finish with it. The only delay introduced by the sampler will be the time to copy, which is expected to be significantly less than the time to process the event data itself.
int get_sample_buffer(base, buffer_lock, copy)
struct pool *base;
    int     buffer_lock;
char    copy[];
{
    int     found;

    /* assume we do not find a buffer to sample */
    found = 0;

    /* check for buffer available for sampling */
    if( base->nsample > 0 )
        {
        /* get exclusive access to shared segment */
        vos_lock(buffer_lock);

        /* recheck for buffer available for sampling */
        if( base->nsample > 0 )
            {
            base->sample_here = (base->sample_here + 1) % nbufs;
            memcpy(copy, base->buffer[base->sample_here], buf_size);
            base->nsample--;    
            found = 1;
            }
        }

    /* release exclusive access to shared segment */
    vos_unlock(buffer_lock);
}

return found;

Before leaving the topic of shared memory, there is an important point to be made about its use. Although there is only one physical memory segment that is being shared simultaneously by several processes, each process may have to refer to this segment with a different virtual address. This can easily occur because when the segment is mapped, vos must find an unused virtual address to map it into, and virtual addresses unused by one process may be used by another, and vice versa. In general, it is impossible to predict which virtual addresses a process has in use, since it depends on the loader, the size of the process's code, stack and data segments, what other shared memory segments may be in use, whatever dynamically allocated memory the process may be using, etc. After all, this difficulty is exactly why we allow the system to pick an unused virtual address when doing the mapping.

The problem that arises from this situation is that the shared memory segment itself should not be used to store addresses, because these have to be virtual addresses and therefore may not be the same for all processes. For example, suppose we wish to construct a linked list in a shared memory segment. Each element of the list is supposed to contain the address of the next element in the list. However, there is no single address for a list element, since this element may be mapped into different virtual addresses for each process accessing it. Therefore, if the process linking an element into the list stored its virtual addresses as pointers, another process trying to use these addresses as pointers would not work properly — it might crash due to an addressing fault, or it might access garbage.

The solution to this problem is to never store addresses in the shared memory segment. If something like an address is needed, as in the linked list example, we should store an
offset relative to the base of the segment, which is computed as: offset = (address − base). Then, when a process wants to use this value as an address, it should first compute: address = offset + base. Since each process has its own unique value of "base", the value of "address" will be the correct virtual address for that process, and the value of "offset" will be the same for all processes.

A similar prohibition exists on the use of shared memory to share any process dependent information. In the producer-consumer example, there is only one mutual exclusion semaphore, but each process must open that semaphore in order to get a handle for it, and the value of this handle will be different for each process (since this handle value is just an arbitrary small integer determined by vos — it has no intrinsic meaning in and of itself). Therefore, we cannot store the handle for the mutual exclusion semaphore in the shared memory, because there is no single handle value — each process has its own, unique handle.

13. SIGNALS

In the producer-consumer example just developed, one process needed to "wait" for a "signal" from another process before it could proceed. This is a very common form of synchronization when programming with more than one process. When the waiting and signaling are as simple as those in our producer-consumer example, they can be accomplished using semaphores. The semaphore must be initialized to the "locked" state. The process waiting for the signal will attempt to "lock" the semaphore, and since it is already locked, the process will block. The process sending the signal will "unlock" the semaphore, causing the blocked process to resume. The order in which the lock and unlock are performed is irrelevant, since if the unlock is performed first, the semaphore will remember this and the process performing the lock will not block but will return at once.

Notice the difference between this use of a semaphore and the previous use. When a semaphore is used to guarantee a process exclusive access to a critical section of code, it is initialized to the unlocked state and it is locked and unlocked by the same process. When a semaphore is used to send signals between two processes, it is initialized to the locked state and one process (the one receiving the signals) always locks the semaphore while the other process (the one sending the signals) always unlocks it.

The producer-consumer example used four routines to accomplish signaling:

setup_signals() to create and initialize the object used for signaling.
endup_signals() to undo the effect of setup_signals().
wait_for_signal() to block the process until the signal arrives.
send_signal() to send the signal.

The implementation of these four routines in terms of semaphores follows. There are two semaphores: one, whose name is given as the value of prodsigname, that is used to send signals from the consumer to the producer; and one, whose name is given as the value of conssigname, that is used to send signals from the producer to the consumer. When a semaphore is first created in vos, it is created in the unlocked state. Therefore, when a semaphore is to be used for signaling, it is must be forced it into the locked state immediately after creation. This is done by simply locking it with vos_lock(). Note that setup_signals() performs all its operations with exclusive access to the shared memory segment in order to guarantee that the process creating these semaphores is also the one to perform the vos_lock() that forces them into the locked state, and to guarantee that the value of signal_connects is updated correctly in the shared memory segment.
void setup_signals(base, buffer_lock, prodh, consh, isproducer)
    struct pool *base;
    int buffer_lock, *prodh, *consh, isproducer;
{
    int status, wascreated;

    /* get exclusive access to shared memory segment */
    vos_lock(buffer_lock);

    /* open semaphore to catch wakeup signals to producer */
    status = vos_open_semaphore(prodh, prodsigname, &wascreated);
    if( vos_failure(status) )
      {
        vos_report_error(status);
        vos_unlock(buffer_lock);
        vos.terminate(0);
      }

    /* initialize newly created semaphore to locked state */
    if( wascreated )
      vos_lock(*prodh);

    /* open semaphore to catch wakeup signals to consumer */
    status = vos_open_semaphore(consh, consigname, &wascreated);
    if( vos_failure(status) )
      {
        vos_report_error(status);
        vos_unlock(buffer_lock);
        vos.terminate(0);
      }

    /* initialize newly created semaphore to locked state */
    if( wascreated )
      vos_lock(*consh);

    /* count number of processes connected to these semaphores */
    base->signal_connects++;

    /* release exclusive access to shared memory segment */
    vos_unlock(buffer_lock);
}

void endup_signals(base, buffer_lock, prodh, consh)
    struct pool *base;
    int prodh, consh;
{

    /* get exclusive access to shared memory segment */
    vos_lock(buffer_lock);

    /* close the semaphores */
    vos_close(prodh);
    vos_close(consh);

    /* reduce number of processes connected to semaphores */
    base->signal_connects--;

    /* if this is last process connected, remove the semaphores */
if( base->signal_connects == 0 )
{
    vos_remove_semaphore(prodsigname);
    vos_remove_semaphore(conssigname);
}

/* release exclusive access to shared memory segment */
vos_unlock(buffer_lock);
}

void wait_for_signal(buffer_lock, wait_object)
int buffer_lock, wait_object;
{
    /* release exclusive access to shared memory segment before waiting */
    vos_unlock(buffer_lock);

    /* wait for other process to send signal */
    vos_lock(wait_object);

    /* get back exclusive access to shared memory segment */
    vos_lock(buffer_lock);
}

void send_signal(signal_object)
int signal_object;
{
    vos_unlock(signal_object);
}

The use of semaphores as a means of signaling has some severe limitations which are overcome by the use of a different type of vos object called the "notification" object. This object is specifically designed for interprocess signaling, and has three advantages over semaphores for this purpose:

1. It is a broadcast device, so that one signal can simultaneously unblock several waiting processes.

2. The wait for arrival of a signal is done using the normal vos_wait_single() and vos_wait_multiple() primitives, so that a timeout can be used, and so that waiting for a signal from another process is treated in the same manner as waiting for a completion signal from an I/O device.

3. An ast routine can be attached to a notification object, that will be activated whenever the signal from another process arrives.

A process creates a notification object by a C statement of the form:

```c
status = vos_open_notification(&handle, name);
```

where:

- `&handle` is the address of an integer into which vos will return a small integer that can be used to reference this notification object in all subsequent vos operations.

- `name` is a string containing the name of a virtual "notification device" in the host system.
The notification object created by a process is mapped onto a notification device in the underlying host system. The function of this device is to receive signals broadcast by other processes, and a process starts the operation of this device, which is to listen for signals, with a C statement of the form:

\[
\text{status} = \text{vos\_start\_listen} (\text{handle, param});
\]

where:

- **handle** is the handle returned by `vos\_open\_notification()`.
- **param** is an arbitrary value that will be passed by vos to any ast routine associated with this object when this operation completes, and that will also be returned from the next wait operation on this object.

When a signal is received, this operation will complete in a manner analogous with the completion of an I/O operation. The process detects this by either waiting on this object, or by attaching an ast routine to this object such that the ast routine will be activated when a signal is received.

The `vos\_open\_notification()` and `vos\_start\_listen()` functions are issued by the process that is to receive the signals. The process that sends the signals does so with a C statement of the form:

\[
\text{status} = \text{vos\_notify} (\text{name});
\]

where:

- **name** is a string whose value is the name of a notification device in the host system.

This statement broadcasts the signal whose name is given by **name** to all processes that have started a "listen" operation on a notification device with this name. The sending process does not itself "open" this device – only the receiving process does this. Signals sent by `vos\_notify()` are not received by processes that have opened a notification device with this name but do not have an outstanding "listen" operation started on it. Receipt of a signal completes the "listen" operation, so that another `vos\_start\_listen()` must be issued in order to receive another signal of the same name.

Using the notification objects instead of semaphores, the four signaling routines from the producer-consumer example can be implemented as follows:

```c
void setup_signals(base, buffer_lock, prodh, consh, isproducer)
{
    struct pool *base;
    int buffer_lock, *prodh, *consh, isproducer;

    int status, wascreated;

    if( isproducer )
    { /* open notification device to catch signals sent to producer */
        status = vos\_open\_notification(prodh, prodsigname);
        if( vos\_failure(status) )
        {
            vos\_report\_error(status);
            vos\_terminate(0);
        }
        *consh = (int)(conssigname);
    } else
```

...
```c
/* open notification device to catch signals sent to consumer */
status = vos_open_notification(consh, conssigname);
if( vos_failure(status) )
{
    vos_report_error(status);
    vos_terminate(0);
}
*prodh = (int)(prodsigname);
}

void endup_signals(base, prodh, consh)
struct pool *base;
int prodh, consh;
{
    /* close the notification objects */
    if( prodh != (int)(conssigname) )
        vos_close(prodh);
    if( consh != (int)(prodsigname) )
        vos_close(consh);
}

void wait_for_signal(buffer_lock, wait_object)
inint buffer_lock, wait_object;
{
    int actual, param;
    /* start notification device so it can receive signals */
    vos_start_listen(wait_object, 0);
    /* release exclusive access to shared memory segment before waiting */
    vos_unlock(buffer_lock);
    /* get back exclusive access to shared memory segment */
    vos_wait_single(wait_object, VOS_NOTIMEOUT, &actual, &param);
    vos_lock(buffer_lock);
}

void send_signal(signal_object)
inint signal_object;
{
    vos_notify((char *)signal_object);
}

14. NETWORKING
The network communication supported by vos is a high-level, message-oriented interface that is relatively independent of the underlying network protocols and transport services of the host systems. It is based on the client-server model of UNIX. There are five types of network objects:
(1) a server listening post, on which the server listens for requests to connect from arbitrary remote clients.
```
(2) a server input channel, on which the server receives messages from a particular connected remote client.

(3) a server output channel, on which the server sends messages to a particular connected remote client.

(4) a client input channel, on which the client receives messages from a remote server.

(5) a client output channel, on which the client sends messages to a remote server.

Items (2) and (3), and items (4) and (5), are inseparably paired, one item in the pair being the input part and the other item the output part of the same network channel. Both objects in a pair are opened simultaneously by a single open function, and both are closed simultaneously by a single call to vos_close().

14.1. Network Servers

A server process begins its network interactions by creating a "listening post" object with a C statement of the form:

```c
status = vos_open_server(&handle, name, protocol);
```

where:

- **&handle** is the address of an integer into which vos will return a small integer that can be used to reference this network object in all subsequent vos operations.
- **name** is a string whose value is the name by which this server wants to be known network-wide.
- **protocol** is a string whose value is the name of the transport service the server wishes to use. At present only two names are known to vos: DECNET and TCPIP. If this parameter is NULL, vos will choose a default transport service according to the value of the environment variable vos_protocol.

If a server wants to be known by several different names, it just opens one listening post object for each name. Once the server has made its name known network-wide, it can start listening for connection requests from remote clients by executing a statement in C of the form:

```c
status = vos_start_listen(handle, param);
```

where:

- **handle** is the handle returned by vos_open_server().
- **param** is an arbitrary value that will be passed by vos to any ast routine associated with this object when this operation completes, and that will also be returned from the next wait operation on this object.

As with all vos_start_zzz(), operations, vos_start_listen() does not wait for receipt of a connection request, but returns immediately, allowing the listen to proceed in parallel. The listen operation will complete when a remote client performs a vos_open_client() specifying the name, node, and protocol of this server process. The server can detect this completion by either waiting on this listening post object, or attaching an ast routine to it that will be activated when the completion occurs. Once a listen operation has completed, another call to vos_start_listen() must be issued in order to receive subsequent connection requests from remote clients.
After the listen request completes, the server process needs to explicitly accept the request and establish a communications channel to the requesting client by executing a statement in C of the form:

```c
status = vos_open_connection(&inhandle, &outhandle, handle);
```

where:

- `&inhandle` is the address of an integer into which vos will return a small integer that can be used to reference an input channel to receive data from the remote client.

- `&outhandle` is the address of an integer into which vos will return a small integer that can be used to reference an output channel to send data to the remote client.

- `handle` is the handle of the listening post object for which the `vos_start_listen()` operation just completed (i.e., the one on which the request for this connection occurred).

This operation creates a pair of network objects, one for input and one for output operations with a remote client. The protocol used by these objects will be the same as the protocol defined by the listening post object on which the connection was requested. If either of the objects in this pair is explicitly closed by calling `vos_close()`, the other object is automatically closed by vos as well. Such a close operation will break the network connection between this server and the remote client.

Communication with the remote client process is accomplished by using the standard vos read and write functions, such as `vos_start_read()`, `vos_start_read_string()`, etc., on the appropriate object. Since each remote client has unique input and output objects, a single server can communicate simultaneously with a large number of client processes, limited only by any bounds imposed by the host system on the number of network connections a single process can have. In addition, the server can simultaneously be listening for new connections on its listening post object.

### 14.2. Network Clients

A client process begins its network transactions by first establishing a connection to a remote server process through the execution of a statement in C of the form:

```c
status = vos_open_client(&inhandle, &outhandle, name, node, protocol);
```

where:

- `&inhandle` is the address of an integer into which vos will return a small integer that can be used to reference an input channel to receive data from the remote server.

- `&outhandle` is the address of an integer into which vos will return a small integer that can be used to reference an output channel to send data to the remote server.

- `name` is a string whose value is the network-wide name of the server to which this client wishes to connect.

- `node` is a string whose value is the network-wide name of the node on which the server resides.
protocol is a string whose value is the name of the transport service the client wishes to use.

This operation succeeds when the remote server issues a vos_open_connection() in response to this request. At that time, vos creates a pair of objects, one for input and one for output, on which the client process can issue the standard vos read and write operations in order to get data from and send data to the remote server. If either of the objects in this pair is explicitly closed by calling vos_close(), the other object is automatically closed by vos as well. Such a close operation will break the network connection between this client and the remote server. Since each remote server has unique input and output objects, a single client can communicate simultaneously with a large number of server processes, limited only by any bounds imposed by the host system on the number of network connections a single process can have.

All communication between a client and a server is in the form of messages. Each message is written into one end of a channel by vos_start_write() or some variation thereof, and it is read from the other end by vos_start_read() or a variant thereof. Message boundaries are preserved regardless of the underlying network protocol. A read from a network input object will return a status of VOS_ENDOFFILE if the remote process disconnects for any reason, including closing its end of the channel, terminating the process, etc.

15. LOGICAL NAMES

It is often necessary for a portable program to be able to pick up information about its environment that will change from one environment to another. This was illustrated in the vos calls for networking, where a process had to know the name of the protocol it should use, and where a client process had to know the name of the node on which a server process resided. Another common example is the name of a file to be used by an application. Not only the value but the syntax of this name as well may be significantly different from one host system to another. It would therefore be virtually impossible to encode such names into a program that was intended to be portable – the program must pick up these names dynamically from the environment in which it is running, and it is this "pick up" which can easily be done in a portable manner by a C statement of the form:

status = vos_translate_logical_name(table, name, buffer, max, &length);

where:

table is a string containing the name of the system table in which to look for the parameter name.

name is a string whose value is the name to be translated.

buffer is a string into which vos will store the translation of name.

max is the maximum number of characters which can be stored in the string buffer.

&length is the address of an integer into which vos will store the actual number of characters in buffer. This value will never be greater than the value of max.

With a tool such as this, the user has only to define the logical name in the host environment before running the program, in whatever syntax is applicable to that host, and the program will then be able to use this primitive to obtain that host-dependent definition in a host independent manner. On UNIX, a logical name is defined by issuing a shell command of the form:
setenv server_node priam

where:

server_node is the logical name.
priam is its definition.

In VMS, the corresponding DCL command would be:

define server_node priam

If a program were run after doing this setenv or define, and it executed the following statement:

status = vos_translate_logical_name(NULL,"server_node", buffer, 100, &length);

then the value stored in buffer would be the 5 character string “priam”, and the value of length would be 5.

16. TIME

In a real-time system, explicit and implicit references to time appear in many different contexts. Some of the requirements are:

(1) The ability to obtain the current calendar day and wall-clock time.
(2) The ability to measure the elapsed time taken between the starting and stopping of a timer.
(3) The ability to sleep (i.e., to release the processor for use by other processes).
(4) The ability to supply timeouts which limit the length of time a process will wait for a signal to arrive.
(5) The ability to create objects, called “alarm clocks”, which will produce signals at fixed time intervals.

The ability to sleep is provided in vos by a C statement of the form:

status = vos_sleep(timeout);

where:

timeout is the maximum number of milliseconds the process is to sleep.

If the sleep lasts for the full timeout milliseconds, the returned status value will be VOS_TIMEOUT. However, if a signal arrives for this process from any source or any device, the ast routine to handle this signal is activated, and then the sleep ends prematurely with a return status value of VOS_NORMAL. This function has already been demonstrated in the section on I/O examples.

The ability to supply timeouts to vos_wait_single() and vos_wait_multiple() has already been discussed and illustrated in the producer-consumer examples.

The ability to measure elapsed time by starting and stopping a timer, and the ability to obtain the current date and time, will be illustrated in the section on performance.

The rest of this section will discuss the “alarm clock” objects. These objects allow a process to send signals to itself at specific time intervals. One of the problems we encounter in designing such a mechanism is the specification of the units in which time is to be measured. For some purposes, a fine-grain resolution on the order of microseconds or even nanoseconds may be appropriate, while for others, resolution on the order of seconds
or minutes is sufficient. To solve this problem, vos allows the user to specify the units in
which he or she wants time measured when the alarm clock object is created. This is done
by a statement in C of the form:

    status = vos_open_alarm_clock(&handle, units, &resolution);

where:

    &handle     is the address of an integer into which vos will return a small integer that
                 can be used to reference this alarm clock object in all subsequent vos
                 operations.
    units        is the units in which the user wishes this clock to measure time,
                 expressed as an exponent of 10, so that, for example, \( -9 \) means
                 nanoseconds (10 to the minus ninth seconds), \( -6 \) means microseconds,
                 \( -3 \) milliseconds, 0 seconds, etc.
    &resolution is the address of an integer into which vos will store the smallest number
                 of the user's units which the host system can record. For example, if
                 units is \( -6 \) (i.e., microseconds) and the host system can only count in
                 units of 10 milliseconds, the value returned in resolution would be
                 10000.

Once an alarm clock object has been created, it is set into operation by a C statement
of the form:

    status = vos_start_alarm(handle, timeout, period, param);

where:

    handle     is the handle returned by vos_open_alarm_clock().
    timeout     is the amount of time, expressed in the user's units, before the clock should
                 issue its first alarm signal.
    period      is the amount of time, expressed in the user's units, between subsequent
                 alarm signals after the first. If only one signal is wanted, this value should
                 be zero.
    param       is an arbitrary value that will be passed by vos to any ast routine associated
                 with this object when this an alarm signal is generated, and that will also be
                 returned from the next wait operation on this object.

The program can detect the arrival of signals generated by an alarm clock by either
explicitly waiting on the alarm clock object, or by attaching an ast routine to this object
that will be activated when they arrive. If the value of the period argument to
vos_start_alarm() is not zero, the alarm clock will generate a sequence of signals that can
be stopped by issuing another vos_start_alarm() on this object, or by a statement in C of
the form:

    status = vos_stop(handle);

where:

    handle     is the handle for the alarm clock object.

A process can have any number of alarm clock objects, each with its own resolution
and periodicity.
17. PERFORMANCE MEASUREMENT AND EVALUATION

One of the goals of vos was to be efficient in real-time. The question therefore arises, how efficient is it? To answer this we designed a series of tests, each designed to measure the performance of a single vos object, such as a semaphore. Each test consists of a set of programs:

- one using the standard vos interface.
- one using the production version of vos.
- one for each host operating system that makes direct calls to that host system rather than going through vos.

The versions of each test program that make direct calls to the host system are, of course, not portable, but must be recoded for each host we are testing. The "standard" version of the vos interface includes a number of error checks on the value of parameters, the sequence of operations, etc., which is useful during the debugging stages of application development, but which imposes a small amount of additional overhead. The "production" version of the vos interface simply eliminates these checks. The user makes no change to his or her program when changing between these two versions, since all the calls and constants remain identical. Rather, the user simply links his or her program with either the standard or production versions of the vos library.

These tests have been run on over 15 hardware and operating system combinations. We attempted to make these runs on an "empty" machine, when no other programs were running, but this is not always possible in a network environment. We recorded their elapsed real-times and then averaged this over several runs.

An example of such a test is the one designed to test semaphore operations. It consists of a single process that repeatedly locks and then unlocks the same semaphore. The code for the vos version of this test, called "semaperf", is:

```c
/* semaperf.c */
/* program to measure performance of vos semaphores */
/* invocation: semaperf [semaphore-name [number]] */

#include <stdio.h>
#include <rdr/vos/vos_cmacros.h>
#include <rdr/vos/vos_def.h>

#define buf_size 256

main(argc, argv)
    int argc;
    char *argv[];
{
    register int   i, k;
    int             n, x, semah, res;
    char            buffer1[buf_size], buffer2[buf_size];
    char            name[buf_size];

    /* start by getting the name of the semaphore */
    if( argc <= 1 )
    {
        printf("enter name of semaphore\n");
        scanf("%s", name);
    }
```
else if (argc <= 3)
    strcpy(name, argv[1]);
else
{
    printf("usage: semaperf [semaphore-name [number]]\n");
    vos_terminate(0);
}

/* now open the semaphore */
if (vos_failure(k=vos_open_semaphore(&semah, name, &res)) )
{
    vos_report_error(k);
    vos_terminate(0);
}

/* semaphore opened ok, now get number of loop iterations */
if (argc != 3)
{
    printf("enter number of lock/unlocks\n");
    scanf("%d", &n);
}
else
    n = atol(argv[2]);

/* all set to go, let I/O die down, then start the test */
vos_sleep(1000);

vos_get_day_and_time(buffer1);
vos_start_timing(-3, &res);
for (i = n; --i >= 0; )
{
    if ( (k=vos_lock(semah)) != VOS_NORMAL )
    {
        vos_report_error(k);
        vos_terminate(0);
    }
    if ( (k=vos_unlock(semah)) != VOS_NORMAL )
    {
        vos_report_error(k);
        vos_terminate(0);
    }
}

vos_stop_timing(&x);
vos_get_day_and_time(buffer2);

printf("semaperf starting: %s\n", buffer1);
printf("semaperf stopping: %s\n", buffer2);
printf(" elapsed time: %f secs\n", ((float)(x))/1000.0);

/* remove semaphore and terminate */
vos_close(semah);
vos_remove_semaphore(name);
vos_terminate(0);
}

The first part of this program consists of logic to pick up the two parameters to its operation: the name of the semaphore and the number of iterations of the test loop. If these parameters are not supplied in the command line, this program explicitly prompts for them
and reads them from the standard input unit. After the semaphore has been opened and everything is ready to go, the process sleeps for a second to let the system get quiet, and then we have the statement:

```c
vos_get_day_and_time(buffer1);
```

which stores into `buffer1` the current date and time as a string similar to:

```
Thu Aug 1 10:34:21 1991
```

The next statement,

```c
vos_start_timing(-3, &res);
```

starts a timer to measure elapsed real-time. The first argument indicates the units in which to measure time as a power of 10, so that the -3 in this case indicates milliseconds. The second argument is the address of an integer into which `vos_start_timing()` stores the resolution of the host system timer in these units. We ignore this result in this test program.

The test loop itself is trivial. We simply lock and then unlock the semaphore n times, where n is one of the parameters set up in the first part of the program. Each lock and unlock operation is tested for success or failure, but, of course, success is expected. As soon as the loop terminates, we have the statement:

```c
vos_stop_timing(&x);
```

which stops the timer started by `vos_start_timing()` and stores into `x` the amount of elapsed time (as the number of milliseconds) since the timer was started. The next statement:

```c
vos_get_day_and_time(buffer2);
```

records the current date and time into `buffer2`. Finally, all the times are printed, the semaphore is closed and removed, and the program terminates.

The program to test semaphore operations directly on each host has the same structure and logic as the one just shown, but all the references to vos routines, in particular `vos_open_semaphore()`, `vos_lock()`, `vos_unlock()`, and `vos_remove_semaphore()`, are replaced by direct calls to the corresponding system routines on that host. The corresponding semaphore test programs for UNIX, called “semaunix”, and for VMS, called “semavms”, follow:

```c
/* semaunix.c */

/* program to measure performance of unix semaphores */
/* same structure as semaperf for VOS semaphores */

#include <stdio.h>
#ifndef sun
#include <time.h>
#else
#include <sys/time.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/file.h>
#endif

#ifndef sgi
if ultrix|AIX|_AIX|SYSTEM_FIVE|apollo
union semun
{
    int val;
    struct semid_ds *buf;
    ushort array[1];
};
#endif
#endif

#define buf_size 256
#define TMP_PATH */tmp/*

union semun arg;
struct sembuf lock_table = (0, -1, SEM_UNDO);
extern time_t time();
extern struct tm *localtime();
extern char *asctime();

void get_day_and_time(string)
    char *string;
{
    time_t x;
    struct tm *local;
    char *t;

    x = time((time_t *)&0);
    local = localtime(&x);
    t = asctime(local);
    strcpy(string, t);
}

struct timeval start_tv, stop_tv;
struct timezone start_tz, stop_tz;

void start_timing()
{
    if( gettimeofday(&start_tv, &start_tz) )
    {
        perror("gettimeofday start");
        exit(-1);
    }
}

void stop_timing(elapsed)
    int *elapsed;
{
    register long val, micro;

    if( gettimeofday(&stop_tv, &stop_tz) )
    {
        perror("gettimeofday stop");
        exit(-1);
    }

    val = (stop_tv.tv_sec - start_tv.tv_sec)*1000;
    if( (micro = stop_tv.tv_usec - start_tv.tv_usec) >= 0 )
        val += (micro + 500)/1000;
    else    val += (micro - 500)/1000;
    *elapsed = val;
}
void main(argc, argv)
  int argc;
  char *argv[];
{
  key_t key;
  register int i, id, flags;
  int n, x;
  char buffer1[buf_size], buffer2[buf_size];
  char nmbuf[buf_size];

  /* start by getting the name of the semaphore */
  if( argc <= 1 )
    {
    printf("enter name of semaphore\n");
    scanf("%s", buffer1);
    
    } else if( argc <= 3 )
    strcpy(buffer1, argv[1]);
  else
    {
    printf("usage: semaunix [semaphore-name [number]]\n");
    exit(-1);
    }

  /* now open the semaphore */
  strcpy(nmbuf, TMP_PATH);
  strcat(nmbuf, buffer1);
  if( access(nmbuf, F_OK) < 0 )
    /* path not accessible, try to create it */
    if( (id=open(nmbuf, O_RDONLY|O_APPEND|O_CREAT, 0666)) < 0 )
      /* nothing works */
      perror("semaphore open");
      exit(-1);
    
    else if (close(id) < 0 )
      {
      perror("semaphore close");
      exit(-1);
      }

  /* we can create/access the special file ok, try to create key */
  if( (key = ftok(nmbuf, 'V')) < 0 )
    /* could not create a key */
    perror("semaphore ftok");
    exit(-1);
  
  flags = 0666;    /* read & write access */
  if( (id=semget(key, 1, flags)) < 0 )
    /* could not access existing semaphore, create it */
    flags |= IPC_CREAT;
    id = semget(key, 1, flags);    /* try again */
  if( id < 0 )
    /* nothing works */
    perror("semaphore semget");
    exit(-1);
  
  /* initial value of newly created semval is 0, change to 1 */
arg.val = 1;
if( semctl(id, 0, SETVAL, arg) )
{
    perror("semaphore semctl SETVAL");
    exit(-1);
}

/* semaphore opened ok, now get number of loop iterations */
if( argc != 3 )
{
    printf("enter number of lock/unlocks\n");
    scanf("%d", &n);
}
else
    n = atol(argv[2]);

/* all set to go, let I/O die down, then start the test */
arg.val = 1;
sleep(1);

get_day_and_time(buffer1);
start_timer();
for( i = n; --i >= 0; )
{
    if( semop(id, &lock_table, 1) < 0 )
    {
        perror("semaphore semop lock");
        exit(-1);
    }
    if( semctl(id, 0, SETVAL, arg) )
    {
        perror("semaphore semctl unlock");
        exit(-1);
    }
}
stop_timer(&x);
get_day_and_time(buffer2);

printf("%s starting: %s", argv[0], buffer1);
printf("%s stopping: %s", argv[0], buffer2);
printf(" elapsed time: %f secs\n", ((float)(x))/1000.0);

/* remove semaphore and terminate */
if( semctl(id, 0, IPC_RMID, 0) < 0 )
(/* could not remove the semaphore */
    perror("semaphore semctl IPC_RMID");
    exit(-1);
)

/* semaphore removed ok, now try to remove the file */
if( unlink(nambuf) < 0 )
(/* could not remove the file */
    perror("semaphore unlink");
    exit(-1);
)

exit(0);
/* semavms.c */

/* program to measure performance of vms semaphores */
/* same structure as semaperf for VOS semaphores */

#include <stdio.h>
#include <time.h>
#include <types.h>
#include <errno.h>

#include <rdr/vos/vos_cm macros.h>
#include <lckdef.h>
#include <psldef.h>
#include <descrip.h>

#define buf_size 256

typedef struct lckblk
{
    unsigned short status;
    unsigned short reserved;
    unsigned int lockid;
    int notify_handle;
} lckblk;

extern struct tm *localtime();
extern char *asctime();

void get_day_and_time(string)
char *string;
{
    time_t x;
    struct tm *local;
    char *t;

    x = time((time_t *)0);
    local = localtime(&x);
    t = asctime(local);
    strcpy(string, t);
}

int main(argc, argv)
int argc;
char *argv[];
{
    register int i;
    int n, x, flags, status;
    char buffer1[buf_size], buffer2[buf_size];
    char nambuf[buf_size];
    lckblk *p;

    $DESCRIPTOR(lockname, "VOS_TIMING");

    /* start by getting the name of the semaphore */
    if( argc <= 1 )
    {
        printf("enter name of semaphore\n");
        scanf("%s", buffer1);
    }
else if( argc <= 3 )
    strcpy(buffer1, argv[1]);
else {
    printf("usage: semavms [semaphore-name [number]]\n");
    exit(-1);
}

/* now open the semaphore */
p = malloc(sizeof(lckblk));

lockname.dsc$a_pointer = buffer1;
lockname.dsc$w_length = (short)strlen(buffer1);
status = sys$enqw(0, LCK$K_NLMODE, p, LCK$M_SYSTEM, &lockname,
    0, 0, 0, 0, PSL$C_USER, 0);
if(!(status % 2))
    lib$signal(status);

/* semaphore opened ok, now get number of loop iterations */
if( argc != 3 )
{
    printf("enter number of lock/unlocks\n");
    scanf("%d", &n);
}
else
    n = atol(argv[2]);

/* all set to go, let I/O die down, then start the test */
sleep(1);

get_day_and_time(buffer1);
vos_start_timing(-3, &flags);
for( i = n; --i >= 0; )
{
    status = sys$enqw(0, LCK$K_EXMODE, p, LCK$M_CONVERT,
        0, 0, 0, 0, PSL$C_USER, 0);
    if(!(status % 2))
        lib$signal(status);

    status = sys$enqw(0, LCK$K_NLMODE, p, LCK$M_CONVERT,
        0, 0, 0, 0, PSL$C_USER, 0);
    if(!(status % 2))
        lib$signal(status);
}
vos_stop_timing(&x);
get_day_and_time(buffer2);

printf("%s starting: %s", argv[0], buffer1);
printf("%s stopping: %s", argv[0], buffer2);
printf("elapsed time: %f secs\n", (float)(x)/1000.0);

/* remove semaphore and terminate */
status = sys$deq(p->lockid, 0, 0, 0);
if(!(status % 2))
    lib$signal(status);

exit(0);
Comparison of these three programs gives an excellent indication of the amount of system dependent detail hidden by vos, as well as the significant differences in detail necessary to perform the same function on different hosts.

Running the first program on a SUN 3/60, a UNIX-based system, with the command:

```
semaperf junky 1000
```

produced the following output:

```
semaperf starting: Thu Aug 1 11:06:36 1991
semaperf stopping: Thu Aug 1 11:06:37 1991
elapsed time: 1.00000000 secs
```

For the command:

```
semaunix junky 1000
```

the output was:

```
semaunix starting: Thu Aug 1 11:10:09 1991
semaunix stopping: Thu Aug 1 11:10:10 1991
elapsed time: 0.98000000 secs
```

The difference in these elapsed times, 0.02 seconds or 2%, is due to the vos overhead. We observed considerable variation in the timings from run to run on the same machine, so we did three things to get a better comparison: we chose the number of loop iterations so that the test would run for at least 10 seconds; we ran the test five to ten times and averaged the observed times; we normalized the average to 100,000 iterations. Table 3 shows the results for the host-specific semaphore test, of which semaunix is the UNIX version:

**Table 3**

Average elapsed times for 100,000 iterations of host-specific semaphore test

<table>
<thead>
<tr>
<th>Host machine</th>
<th>Operating System</th>
<th>Host-specific time (secs)</th>
<th>Speed ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC 3100</td>
<td>ULTRIX 3.1</td>
<td>9.817</td>
<td>15.23</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>AIX</td>
<td>11.928</td>
<td>12.54</td>
</tr>
<tr>
<td>DEC 5000</td>
<td>ULTRIX 4.0</td>
<td>13.864</td>
<td>10.79</td>
</tr>
<tr>
<td>Silicon Graphics</td>
<td>IRIX 3.3</td>
<td>16.979</td>
<td>8.81</td>
</tr>
<tr>
<td>DEC 3100</td>
<td>ULTRIX 4.1</td>
<td>20.395</td>
<td>7.34</td>
</tr>
<tr>
<td>VAX 3900</td>
<td>VMS 5.3</td>
<td>22.650</td>
<td>6.60</td>
</tr>
<tr>
<td>MODCOMP</td>
<td></td>
<td>26.000</td>
<td>5.75</td>
</tr>
<tr>
<td>DEC 2100</td>
<td>ULTRIX 4.0</td>
<td>26.557</td>
<td>5.63</td>
</tr>
<tr>
<td>SUN SLC</td>
<td>SunOS 4.1</td>
<td>28.460</td>
<td>5.25</td>
</tr>
<tr>
<td>Apollo DN10k</td>
<td>SunOS 4.0.3c</td>
<td>29.517</td>
<td>5.06</td>
</tr>
<tr>
<td>SUN Sparc1</td>
<td>SunOS 4.0.3c</td>
<td>33.844</td>
<td>4.42</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>VMS 5.3</td>
<td>34.590</td>
<td>4.32</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>ULTRIX-32 3.0</td>
<td>42.561</td>
<td>3.52</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>ULTRIX 4.1</td>
<td>64.380</td>
<td>2.32</td>
</tr>
<tr>
<td>SUN 3/60</td>
<td>SunOS 4.1.1</td>
<td>98.280</td>
<td>1.52</td>
</tr>
<tr>
<td>SUN 3/60</td>
<td>SunOS 4.0.3</td>
<td>112.800</td>
<td>1.33</td>
</tr>
<tr>
<td>SUN 3/75</td>
<td>SunOS 4.1.1</td>
<td>121.360</td>
<td>1.23</td>
</tr>
<tr>
<td>µVAX II</td>
<td>ULTRIX 4.0</td>
<td>149.610</td>
<td>1.00</td>
</tr>
</tbody>
</table>
The column labeled "Speed ratio" is the ratio of the elapsed time of the host-specific test for this machine to the elapsed time for the same test on the microVAX II, the slowest machine tested. The table is ordered by decreasing values of this ratio. A notable result is the effect which the operating system can have on performance. The DEC 3100 was tested at 9.817 seconds when running semaunix on the ULTRIX 3.1 operating system, but when this same test was run on the same machine with ULTRIX 4.1, it took 20.395 seconds, a factor of 2.1 slower! A similar but less dramatic effect occurred on the VAX 3100: running on ULTRIX-32 3.0, semaunix took 42.561 seconds, but running on ULTRIX 4.1, the same test took 64.380 seconds, a factor of 1.5 slower. Such results seem to indicate that the newer version of ULTRIX has seriously compromised the speed of its semaphore operations, independently of the hardware on which it runs (DEC 3100 or VAX 3100).

However, not all newer releases of an operating system cause performance degradation. The SUN 3/60 running SunOS 4.0.3 required 112.8 seconds to run semaunix. When run with SunOS 4.1.1, the same test on the same hardware required only 98.28 seconds.

Table 4 shows the overhead in the production version of vos, calculated as the difference in the average elapsed times for semaperf and semaunix. For most machines, this is on the order of a few percent, although it is quite high for the VAX and SPARC architectures, perhaps due to the expense of procedure calls on those machines. The table entries are ordered by increasing percentage of overhead, as given in the last column.

**Table 4**

<table>
<thead>
<tr>
<th>Host Machine</th>
<th>Operating system</th>
<th>Host specific time (secs)</th>
<th>Semaperf production time (secs)</th>
<th>Semaperf production overhead (secs) (percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN 3/60</td>
<td>SunOS 4.1.1</td>
<td>98.280</td>
<td>99.640</td>
<td>1.360 (1.38%)</td>
</tr>
<tr>
<td>SUN 3/60</td>
<td>SunOS 4.0.3</td>
<td>112.800</td>
<td>114.810</td>
<td>2.010 (1.78%)</td>
</tr>
<tr>
<td>SUN 3/75</td>
<td>SunOS 4.1.1</td>
<td>121.360</td>
<td>123.920</td>
<td>2.560 (2.11%)</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>AIX</td>
<td>11.928</td>
<td>12.198</td>
<td>0.270 (2.26%)</td>
</tr>
<tr>
<td>DEC 5000</td>
<td>ULTRIX 4.0</td>
<td>13.864</td>
<td>14.258</td>
<td>0.394 (2.84%)</td>
</tr>
<tr>
<td>Apollo DN10k</td>
<td>ULTRIX 4.0</td>
<td>29.517</td>
<td>30.701</td>
<td>1.184 (4.01%)</td>
</tr>
<tr>
<td>DEC 3100</td>
<td>ULTRIX 4.1</td>
<td>20.395</td>
<td>21.245</td>
<td>0.850 (4.17%)</td>
</tr>
<tr>
<td>MODCOMP</td>
<td>ULTRIX 4.1</td>
<td>26.000</td>
<td>27.533</td>
<td>1.535 (5.89%)</td>
</tr>
<tr>
<td>μVAX II</td>
<td>ULTRIX 4.0</td>
<td>149.610</td>
<td>161.200</td>
<td>11.590 (7.75%)</td>
</tr>
<tr>
<td>DEC 2100</td>
<td>ULTRIX 4.0</td>
<td>26.557</td>
<td>28.948</td>
<td>2.391 (9.00%)</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>ULTRIX 4.1</td>
<td>64.380</td>
<td>70.238</td>
<td>5.858 (9.10%)</td>
</tr>
<tr>
<td>Silicon Graphics</td>
<td>IRIX 3.3</td>
<td>16.979</td>
<td>18.991</td>
<td>2.012 (11.84%)</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>ULTRIX-32 3.0</td>
<td>42.561</td>
<td>48.310</td>
<td>5.749 (13.51%)</td>
</tr>
<tr>
<td>SUN Sparc1</td>
<td>SunOS 4.0.3c</td>
<td>33.844</td>
<td>38.936</td>
<td>5.092 (15.04%)</td>
</tr>
<tr>
<td>DEC 3100</td>
<td>ULTRIX 3.1</td>
<td>9.817</td>
<td>11.421</td>
<td>1.604 (16.33%)</td>
</tr>
<tr>
<td>VAX 3900</td>
<td>VMS 5.3</td>
<td>22.650</td>
<td>26.630</td>
<td>3.980 (17.57%)</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>VMS 5.3</td>
<td>34.590</td>
<td>40.850</td>
<td>6.260 (18.09%)</td>
</tr>
<tr>
<td>SUN SLC</td>
<td>SunOS 4.1</td>
<td>28.460</td>
<td>35.590</td>
<td>7.130 (25.05%)</td>
</tr>
</tbody>
</table>

Table 5 shows the same values as Table 4, but grouped according to the six underlying CPU chips of the hosts used in these tests. In general, the overhead is highest on VAX and SPARC, lowest on the Motorola 68000, PRISM, and IBM RISC, and moderate on the MIPS. This is independent of the operating system in use on the VAX (which was tested with VMS and two versions of ULTRIX), so it would appear that the overhead is due to the hardware instruction set, most likely the expense incurred in procedure entry.
Table 5
Vos production overhead grouped by chip families

<table>
<thead>
<tr>
<th>Host Machine</th>
<th>Operating system</th>
<th>Host specific time (secs)</th>
<th>Semaperf production time (secs)</th>
<th>Semaperf production overhead (secs)</th>
<th>(percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN 3/60</td>
<td>SunOS 4.1.1</td>
<td>98.280</td>
<td>99.640</td>
<td>1.360</td>
<td>1.38%</td>
</tr>
<tr>
<td>SUN 3/60</td>
<td>SunOS 4.0.3</td>
<td>112.800</td>
<td>114.810</td>
<td>2.010</td>
<td>1.78%</td>
</tr>
<tr>
<td>SUN 3/75</td>
<td>SunOS 4.1.1</td>
<td>121.360</td>
<td>123.920</td>
<td>2.560</td>
<td>2.11%</td>
</tr>
<tr>
<td>MODCOMP</td>
<td></td>
<td>26.000</td>
<td>27.533</td>
<td>1.535</td>
<td>5.89%</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>AIX</td>
<td>11.928</td>
<td>12.198</td>
<td>0.270</td>
<td>2.26%</td>
</tr>
<tr>
<td>Apollo DN10k</td>
<td></td>
<td>29.517</td>
<td>30.701</td>
<td>1.184</td>
<td>4.01%</td>
</tr>
<tr>
<td>DEC 5000</td>
<td>ULTRIX 4.0</td>
<td>13.864</td>
<td>14.258</td>
<td>0.394</td>
<td>2.84%</td>
</tr>
<tr>
<td>DEC 3100</td>
<td>ULTRIX 4.1</td>
<td>20.395</td>
<td>21.245</td>
<td>0.850</td>
<td>4.17%</td>
</tr>
<tr>
<td>DEC 2100</td>
<td>ULTRIX 4.0</td>
<td>26.577</td>
<td>28.948</td>
<td>2.391</td>
<td>9.00%</td>
</tr>
<tr>
<td>Silicon Graphics</td>
<td>IRIX 3.3</td>
<td>16.979</td>
<td>18.991</td>
<td>2.012</td>
<td>11.84%</td>
</tr>
<tr>
<td>DEC 3100</td>
<td>ULTRIX 3.1</td>
<td>9.817</td>
<td>11.421</td>
<td>1.604</td>
<td>16.33%</td>
</tr>
<tr>
<td>μVAX II</td>
<td>ULTRIX 4.0</td>
<td>149.610</td>
<td>161.200</td>
<td>11.590</td>
<td>7.75%</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>ULTRIX 4.1</td>
<td>64.380</td>
<td>70.238</td>
<td>5.858</td>
<td>9.10%</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>ULTRIX-32 3.0</td>
<td>42.561</td>
<td>48.310</td>
<td>5.749</td>
<td>13.51%</td>
</tr>
<tr>
<td>VAX 3900</td>
<td>VMS 5.3</td>
<td>22.650</td>
<td>26.630</td>
<td>3.980</td>
<td>17.57%</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>VMS 5.3</td>
<td>34.590</td>
<td>40.850</td>
<td>6.260</td>
<td>18.09%</td>
</tr>
<tr>
<td>SUN Sparc1</td>
<td>SunOS 4.0.3c</td>
<td>33.844</td>
<td>38.936</td>
<td>5.092</td>
<td>15.04%</td>
</tr>
<tr>
<td>SUN SLC</td>
<td>SunOS 4.1</td>
<td>28.460</td>
<td>35.590</td>
<td>7.130</td>
<td>25.05%</td>
</tr>
</tbody>
</table>

Table 6 shows the additional overhead attributable to the extra checking performed by the standard version of vos as compared to the production version. The comparison, done on only three machines, shows an increase in overhead on the order of one or two percent.

Table 6
Additional overhead of standard vos as measured by semaperf

<table>
<thead>
<tr>
<th>Host Machine</th>
<th>Operating system</th>
<th>Semaperf production time (secs)</th>
<th>Semaperf standard time (secs)</th>
<th>Additional overhead due to checking (secs)</th>
<th>(percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN 3/60</td>
<td>SunOS 4.0.3</td>
<td>114.810</td>
<td>115.600</td>
<td>0.790</td>
<td>0.69%</td>
</tr>
<tr>
<td>VAX 3100</td>
<td>ULTRIX-32 3.0</td>
<td>48.310</td>
<td>48.811</td>
<td>0.501</td>
<td>1.04%</td>
</tr>
<tr>
<td>μVAX II</td>
<td>ULTRIX 4.0</td>
<td>161.200</td>
<td>164.000</td>
<td>2.800</td>
<td>1.74%</td>
</tr>
</tbody>
</table>

18. IMPLEMENTATION

Comparison of the code for the programs semaperf,_semaunix, and semavms, as given in the section on performance evaluation, gives a good indication of how vos semaphores are implemented on UNIX and VMS systems. On OS-9 system, semaphores are implemented using events. The details on all three systems are completely different, yet by hiding all these details, vos presents a uniform interface to application programs that makes their portability relatively simple.
Difficulties may be encountered in this portability when a particular host imposes restrictions on its implementation of vos. For example, in UNIX, semaphore names are used by vos as UNIX file names, and are therefore limited to 255 characters, which is not a serious limitation. In VMS, semaphore names are used by vos as VMS resource names, and are therefore limited to 31 characters. In OS-9, semaphore names are used as OS-9 event names, which are limited to just eleven characters, a very serious limitation. In order to be maximally portable, a program must confine itself to the limits of the most restrictive system, which in this case would mean limiting semaphore names to just eleven characters.

One simple way around this difficulty with names would be to use a logical name to "pick up" the actual semaphore name from the host environment by calling \texttt{vos\_translate\_logical\_name()}. The person running such a program on any particular host could then supply a name consistent with the syntax and restrictions on that host. The program could be written such that if \texttt{vos\_translate\_logical\_name()} fails (i.e., if the logical name is not defined in the host environment), it would default to something suitable for the most restrictive system, in this case an 11-character name.

Vos is implemented in two languages: C and Fortran, although the bulk of it is written in C. As mentioned in the section on motivation, we implemented many of the primitives by first simulating POSIX primitives and then using those internally. In addition, the CATS networking interface was also integrated into vos in order to accommodate existing programs already using that interface.

Table 7 shows the number of lines of source code needed to implement vos. There are four parts to this table. The first shows the number of lines of source code that is itself machine independent and that is included in the vos implementation on all host systems. The second, third, and fourth parts show the number of lines of code that is specific to UNIX, OS-9, and VMS respectively. An implementation on one of those hosts would obviously only utilize the source code for that host in addition to the independent code used on all hosts. As can be seen for each of the three implementations to date, roughly 50% of the source code is unique to a particular host, and 50% is independent of that host. If vos were to be ported to another operating system, a new version of the machine dependent code would have to be written for that system, using the three existing versions as a guide, but the machine independent code should be directly usable without change on the new host.
Table 7
Lines of Source Code in VOS

<table>
<thead>
<tr>
<th>Modules</th>
<th>Language</th>
<th>Number of Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Independent Source Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POSIX</td>
<td>C</td>
<td>635</td>
</tr>
<tr>
<td>CATS</td>
<td>C</td>
<td>1016</td>
</tr>
<tr>
<td>VOS</td>
<td>C</td>
<td>2389</td>
</tr>
<tr>
<td>VOS</td>
<td>FORTRAN</td>
<td>143</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>4183</td>
</tr>
<tr>
<td>Unix Dependent Source Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POSIX</td>
<td>C</td>
<td>1016</td>
</tr>
<tr>
<td>VOS</td>
<td>C</td>
<td>2978</td>
</tr>
<tr>
<td>VOS</td>
<td>FORTRAN</td>
<td>134</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>4128</td>
</tr>
<tr>
<td>OS-9 Dependent Source Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POSIX</td>
<td>C</td>
<td>850</td>
</tr>
<tr>
<td>VOS</td>
<td>C</td>
<td>2150</td>
</tr>
<tr>
<td>VOS</td>
<td>FORTRAN</td>
<td>101</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>3101</td>
</tr>
<tr>
<td>VMS Dependent Source Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POSIX</td>
<td>C</td>
<td>629</td>
</tr>
<tr>
<td>VOS</td>
<td>C</td>
<td>3724</td>
</tr>
<tr>
<td>VOS</td>
<td>FORTRAN</td>
<td>270</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>4623</td>
</tr>
</tbody>
</table>

The lines labeled "POSIX" in Table 7 implement a simulation of some of the real-time extensions to POSIX. This is based on Draft 9 (December, 1989) of POSIX 1003.4. A summary of those features actually implemented is shown in Table 8. These include asynchronous I/O with the *aioch* asynchronous I/O control block structure, asynchronous event notification with the *event* structure, and clocks and timers.

To port vos to a system that complies with the POSIX real-time extensions, we would have to eliminate the code that simulates this functionality, and implement the vos semaphore and shared memory primitives in terms of the POSIX functions (as host-dependent code). This will clearly lead to a net reduction in the number of lines of code needed to implement vos, and should also lead to an improvement in its performance. We have not yet had the opportunity to try this because manufacturers are only now beginning to supply operating systems that offer the POSIX 1003.4 real-time features.
Table 8

POSIX Real-time Extensions implemented for VOS

<table>
<thead>
<tr>
<th>POSIX Real-time Feature</th>
<th>Routines Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous I/O</td>
<td>aread()</td>
</tr>
<tr>
<td></td>
<td>awrite()</td>
</tr>
<tr>
<td></td>
<td>acancel()</td>
</tr>
<tr>
<td>Clocks and Timers</td>
<td>mktimer()</td>
</tr>
<tr>
<td></td>
<td>rmtimer()</td>
</tr>
<tr>
<td></td>
<td>gettimer()</td>
</tr>
<tr>
<td></td>
<td>reltimer()</td>
</tr>
<tr>
<td></td>
<td>abstimer()</td>
</tr>
<tr>
<td></td>
<td>restrel()</td>
</tr>
<tr>
<td></td>
<td>nanosleep()</td>
</tr>
<tr>
<td>Asynchronous Event Notification</td>
<td>evemptyset()</td>
</tr>
<tr>
<td></td>
<td>evtfi1set()</td>
</tr>
<tr>
<td></td>
<td>evtaddset()</td>
</tr>
<tr>
<td></td>
<td>evtdelset()</td>
</tr>
<tr>
<td></td>
<td>evtsmember()</td>
</tr>
<tr>
<td></td>
<td>evtprocmask()</td>
</tr>
<tr>
<td></td>
<td>evtpoll()</td>
</tr>
<tr>
<td></td>
<td>evtraise()</td>
</tr>
<tr>
<td></td>
<td>evtsuspend()</td>
</tr>
</tbody>
</table>

19. CONCLUSION

Vos has proven to be an effective system for writing portable real-time software. It was first used in a production run during the spring of 1990 by the WA89 experiment at CERN, where it enabled experimenters to integrate DECstations running ULTRIX into a network of VAXes running VMS. Since then, a number of MODEL modules have been rewritten to use vos, including: OSP, the Occurrence Signal Processor; MHI, the MODEL Human Interface; MBM, the MODEL Buffer Manager; and MRS, the MODEL Recording System. Numerous experiments at CERN now use these modules on VMS, OS-9, and UNIX systems of all types. Many users previously using MODEL modules on VMS systems have also converted to the new modules based on vos. In the case of OSP, this led to a significant performance improvement due to the consistent use of asynchronous I/O in vos. Vos has also been used to port off-line software that uses shared memory and semaphores.

The original UNIX version of vos was developed on ULTRIX, but this has since been ported to a number of other UNIX-based platforms. This port is particularly easy, typically taking only a few hours to transfer the source files, recompile and relink them. The only difficulties occur with #ifdef and #include lines in the vos source, because each version of UNIX seems to have its own idea of which files need to be included to define which macro symbols. In addition to everything else, vos provides a convenient mechanism for migrating applications between the various flavors of UNIX!

Acknowledgements

I wish to thank G. Mornacchi, coauthor of vos, and the other members of the OC Group at CERN for their many contributions to the design and development of vos.
REFERENCES


High Performance CISC, RISC and VLIW Processors

G. S. Rao

IBM Data Systems Division, Poughkeepsie, NY, USA

ABSTRACT

RISC, CISC and VLIW represent different design philosophies for achieving high performance computations in processors. These are illustrated using case studies: the IBM ES/9000 for CISC, the IBM RS/6000 for RISC, and the Cydrome Cydra"5 for VLIW.

1. INTRODUCTION

Computer architecture evolution is driven by the market needs it serves and the technology that supports it. In the 60's, architecture and design were driven primarily by hardware technology. As the investment in the programs for these early architectures grew, instruction set compatibility became a major requirement and had a significant influence on the design direction as well. The resulting CISC processors tended to have more complex design points in an effort to achieve high performance without the benefit of knowing how individual programs were making use of the computational resources of the machine.

2. RISC, VLIW, SUPERSCALAR AND CISC PROCESSORS EVOLUTION

In the mid 70's an alternate approach to computing was developed [1] that subsequently came to be called RISC. This shifted the balance of hardware and compiler complexities away from hardware. The essential elements of that approach appear to be: a hardware design that implements only the core set of operations or functions in hardware to simplify it; reliance on a large set of registers and an optimizing compiler for reduced path lengths through scheduling of hardware resources; and separate instruction and data caches for high bandwidth. Advances in compiler optimization [1] made this possible.

The RISC approach of instruction resource scheduling by an optimizing compiler was extended in the 80's in VLIW architecture machines for exploiting fine grain parallelism: first by a compiler that developed likely execution paths for the code and then compacted them into long instruction words [2]; and subsequently by a compiler that developed schedules of a hypothetical dataflow machine [3].

Supported by advances in CMOS technology and design techniques, superscalar designs that extended RISC principles to support multiple instruction dispatch, concurrent operation of multiple functional units emerged in the late 80's [4].

Most recently, traditional CISC instruction set processors supporting superscalar design approaches for high performance have emerged [5], [6]. They exploit advances in ECL technology with a design that permits much higher levels of concurrency of operations, out-of-sequence instruction execution and two levels of cache, and yet maintain upward compatibility of instruction sets.

* Trademark of International Business Machines Corporation.
** Trademark of Cydrome Inc.
REFERENCES


LIST OF PARTICIPANTS

LECTURERS

K.-H. BECKS, Bergische Universität Wuppertal, Germany
G. FLÜGGE, III. Physikalisches Institut der RWTH Aachen, Germany
C.N.P. GEE, Rutherford Appleton Laboratory, Chilton, Didcot, U.K.
J.R. HANSEN, Niels Bohr Institute, Copenhagen, Denmark
D. KEMP, CERN, Geneva, Switzerland
P.F. KUNZ, SLAC, Stanford, U.S.A.
L. MAPELLI, CERN, Geneva, Switzerland
T. NASH, FNAL, Batavia, U.S.A.
D. PARKINSON, DAP Unit, Computing Centre, Queen Mary College, London, U.K.
R. PENROSE, Mathematical Institute, Oxford University, U.K.
C. PETERSON, Lund University, Sweden
G.S. RAO, IBM Data Systems Division, Poughkeepsie, U.S.A.
R.D. RUSSELL, University of New Hampshire, Durham, U.S.A.
R. SHEPHERD, INMOS Ltd., Bristol, U.K.
T. SJÖSTRAND, CERN, Geneva, Switzerland

TUTORIAL SESSIONS

T. RÖGNVALDSSON, Lund University, Sweden
C. JACOBSSON, Lund University, Sweden
M. NYBERG, Lund University, Sweden
STUDENTS

A. Almar, CERN, Geneva, Switzerland
C. Alippi, Politecnico di Milano/Italy and University College London/U.K.
M. Ballintijn, Nikhef, Amsterdam, Netherlands
   (currently at CERN, PPE Division)
P. Baltes, Technische Hochschule Darmstadt, Germany
R. Barillere, CERN, Geneva, Switzerland
F. Block, Bergische Universität Wuppertal, Germany
B.P. Brandl, Heidelberg University, Germany
M. Colombo, Dortmund University c/o DESY, Hamburg, Germany
D. Col.tuc, Research Institute for Electrical Engineering, Bucarest, Romania
M. Dahlinger, GSI, Darmstadt, Germany
S. DeniziaK, Kielce University of Technology, Poland
M. Donszelmann, Nikhef-H, Amsterdam, Netherlands
   (currently at CERN, PPE/DELPHI)
A. Engelhardt, Niels Bohr Institute, Copenhagen, Denmark
H. EsseL, GSI, Darmstadt, Germany
M.R. Fausey, Fermilab, Batavia, U.S.A.
M. Feuerstack, Heidelberg University, Germany
K. Furnival, Oliver Lodge Laboratory, Liverpool University, U.K.
   (currently at CERN, PPE/DELPHI)
H. Goring, GSI, Darmstadt, Germany
J.-J. Gras, CERN, Geneva, Switzerland
C. GuicheneY, Lab. de Physique Corpusculaire, IN2P3, Aubiere, France
P. Hagen, CERN, Geneva, Switzerland
S.J. Haywood, CERN, Geneva, Switzerland
H. Heukenkamp, Institut für Hochenergiephysik, Zeuthen, Germany
J. Hiltgarten, CERN, Geneva, Switzerland
B. Hoffmann, Institut für Hochenergiephysik, Zeuthen, Germany
F. Hoffmann, Hahn-Meitner-Institut, Berlin, Germany
H. Ihsen, Heidelberg University, Germany
C. Joos, Freiburg University, Germany
P. Kubinec, Comenius University, Bratislava, Czechoslovakia
J. Kyriadopoulos, Fermilab, Batavia, U.S.A.
G. Levi, INFN, Perugia, Italy
   (currently at CERN, PPE Division)
T. Linden, Helsinki University, Finland
H. Lothmander, Lund University, Sweden
G. Maelhum, Oslo University, Norway
J. MAINUSCH, DESY, Hamburg, Germany
J. MASARIK, Comenius University, Bratislava, Czechoslovakia
M. MARINO, World Laboratory, Lausanne and CERN, Geneva, Switzerland
J. MEYER, ESRF, Grenoble, France
P. MORETTINI, INFN, Genoa, Italy
A. MULLER, Max-Planck-Institut für Plasmaphysik, Munich, Germany
T. ODEGAARD, Oslo University, Norway
S. OHLSSON, CERN, Geneva, Switzerland
T. OSTROWSKI, Technical University of Szczecin, Poland
S.U. PANDEY, Pittsburgh University, U.S.A.
(currently at CERN, PPE Division)
A.C. PARDE TORRES, Centre de Calcul de l’IN2P3, Villeurbanne, France
A. PERUS, LAL, Orsay, France
H.J. PETERS, DESY -R-, Hamburg, Germany
R. PIETRAK, Warsaw University, Poland
J.B. PURVIS, CERN, Geneva, Switzerland
S. RAVNDAL, Ruhr Universität Bochum, Germany
F. RICCARDI, INFN Napoli, Italy
T. ROVELLI, Universita degli studi di Bologna, Italy
C. SAENZ GAMASA, Universidad de Zaragoza, Spain
E. SCHAFER, Johannes Gutenberg Universität, Mainz, Germany
(currently at CERN, PPE Division)
T. SCHWAB, GSI, Darmstadt, Germany
D. SLIMMER, Fermilab, Batavia, U.S.A.
L. SMOLIK, DESY -FHIT-, Hamburg, Germany
J.B. STAV, Trondheim University, Norway
F. STICOZZI, CALTECH, Pasadena, U.S.A.
(currently at CERN, PPE Division)
G. TOMASICCHIO, INFN, Bari, Italy
H.J. TORGERSEN, Oslo University, Norway
E. TRASATTI, INFN, Rome, Italy
E.E. VEITCH, CERN, Edinburgh University, U.K.
(currently at CERN, PPE Division)
L. VERMEERSCH, Ghent University, Belgium
S.M. WALTHER, Johannes Gutenberg Universität, Mainz, Germany
J. WROLDSEN, Gjovik College of Engineering, Norway
G. XESEO, CERN, Geneva Switzerland
(also affiliated to COPPE/UFRJ, Brazil and World Laboratory, Switzerland)
LIST OF PARTICIPANTS

LECTURERS

K.-H. BECKS, Bergische Universität Wuppertal, Germany
G. FLÜGGE, III. Physikalisches Institut der RWTH Aachen, Germany
C.N.P. GEE, Rutherford Appleton Laboratory, Chilton, Didcot, U.K.
J.R. HANSEN, Niels Bohr Institute, Copenhagen, Denmark
D. KEMP, CERN, Geneva, Switzerland
P.F. KUNZ, SLAC, Stanford, U.S.A.
L. MAPELLI, CERN, Geneva, Switzerland
T. NASH, FNAL, Batavia, U.S.A.
D. PARKINSON, DAP Unit, Computing Centre, Queen Mary College, London, U.K.
R. PENROSE, Mathematical Institute, Oxford University, U.K.
C. PETERSON, Lund University, Sweden
G.S. RAO, IBM Data Systems Division, Poughkeepsie, U.S.A.
R.D. RUSSELL, University of New Hampshire, Durham, U.S.A.
R. SHEPHERD, INMOS Ltd., Bristol, U.K.
T. SjÖSTRAND, CERN, Geneva, Switzerland

TUTORIAL SESSIONS

T. RÖGNVALDSSON, Lund University, Sweden
C. JACOBSSON, Lund University, Sweden
M. NYBERG, Lund University, Sweden
STUDENTS

A. AILMAR, CERN, Geneva, Switzerland
C. ALIPPI, Politecnico di Milano/Italy and University College London/U.K.
M. BALLINTJN, NIKHEF, Amsterdam, Netherlands
   (currently at CERN, PPE Division)
P. BALTES, Technische Hochschule Darmstadt, Germany
R. BARILLERE, CERN, Geneva, Switzerland
F. BLOCK, Bergische Universität Wuppertal, Germany
B.P. BRANDL, Heidelberg University, Germany
M. COLOMBO, Dortmund University c/o DESY, Hamburg, Germany
D. COLTUC, Research Institute for Electrical Engineering, Bucharest, Romania
M. DAHLINGER, GSI, Darmstadt, Germany
S. DENIZIAK, Kielce University of Technology, Poland
M. DONSZELMANN, NIKHEF-H, Amsterdam, Netherlands
   (currently at CERN, PPE/Delphi)
A. ENGELHARDT, Niels Bohr Institute, Copenhagen, Denmark
H. ESSEL, GSI, Darmstadt, Germany
M.R. FAUSEY, Fermilab, Batavia, U.S.A.
M. FEUERSTACK, Heidelberg University, Germany
K. FURNIVAL, Oliver Lodge Laboratory, Liverpool University, U.K.
   (currently at CERN, PPE/Delphi)
H. GORINGER, GSI, Darmstadt, Germany
J.-J. GRAS, CERN, Geneva, Switzerland
C. GUICHENEO, Lab. de Physique Corpusculaire, IN2P3, Aubiere, France
P. HAGEN, CERN, Geneva, Switzerland
S.J. HAYWOOD, CERN, Geneva, Switzerland
H. HEUKENKAMP, Institut für Hochenergiephysik, Zeuthen, Germany
J. HILGART, CERN, Geneva, Switzerland
B. HOFFMANN, Institut für Hochenergiephysik, Zeuthen, Germany
F. HOFFMANN, Hahn-Meitner-Institut, Berlin, Germany
H. IHSSSEN, Heidelberg University, Germany
C. IOOS, Freiburg University, Germany
P. KUBINEC, Comenius University, Bratislava, Czechoslovakia
J. KYRIAKOPULOS, Fermilab, Batavia, U.S.A.
G. LEVI, INFN, Perugia, Italy
   (currently at CERN, PPE Division)
T. LINDEM, Helsinki University, Finland
H. LOHMANDER, Lund University, Sweden
G. MAEHLUM, Oslo University, Norway
J. MAINUSCH, DESY, Hamburg, Germany
J. MASARÍK, Comenius University, Bratislava, Czechoslovakia
M. MARINO, World Laboratory, Lausanne and CERN, Geneva, Switzerland
J. MEYER, ESRF, Grenoble, France
P. MORETTINI, INFN, Genua, Italy
A. MULLER, Max-Planck-Institut für Plasmaphysik, Munich, Germany
T. ODEGAARD, Oslo University, Norway
S. OHLSSON, CERN, Geneva, Switzerland
T. OSTROWSKI, Technical University of Szczecin, Poland
S.U. PANDEY, Pittsburgh University, U.S.A.
(currently at CERN, PPE Division)
A.C.. PARDE TORRES, Centre de Calcul de l’IN2P3, Villeurbanne, France
A. PERUS, LAL, Orsay, France
H.J. PETERS, DESY -R-, Hamburg, Germany
R. PIETRAK, Warsaw University, Poland
J.B. PURVIS, CERN, Geneva, Switzerland
S. RAVNDAL, Ruhr Universität Bochum, Germany
F. RICCARDI, INFN Napoli, Italy
T. ROVELLI, Universita degli studi di Bologna, Italy
C. SAENZ GAMASA, Universidad de Zaragoza, Spain
E. SCHAFFER, Johannes Gutenberg Universität, Mainz, Germany
(currently at CERN, PPE Division)
T. SCHWAB, GSI, Darmstadt, Germany
D. SLIMMER, Fermilab, Batavia, U.S.A.
L. SMOLIK, DESY -FHI-, Hamburg, Germany
J.B. STAV, Trondheim University, Norway
F. STICOZZI, CALTECH, Pasadena, U.S.A.
(currently at CERN, PPE Division)
G. TOMASICCHIO, INFN, Bari, Italy
H.J. TORGersen, Oslo University, Norway
E. TRASATTI, INFN, Rome, Italy
E.E. VEITCH, CERN, Edinburgh University, U.K.
(currently at CERN, PPE Division)
L. VERMEERSCH, Ghent University, Belgium
S.M. WALTHER, Johannes Gutenberg Universität, Mainz, Germany
J. WROLDSEN, Gjøvik College of Engineering, Norway
G. XEXEO, CERN, Geneva Switzerland
(also affiliated to COPPE/UFRJ, Brazil and World Laboratory, Switzerland)