EUROPEAN ORGANISATION FOR NUCLEAR RESEARCH

CERN/DD/89/19
LAA/RT/89/02

*Simulation of a macro-pipelined multi-cpu
event processor for use in FASTBUS*

*M. F. Letheren, CERN/DD
A. Marchioro, CERN/EF
F. Slorach, Rutherford Appleton Laboratory*

CERN LIBRARIES, GENEVA

CM-P00059919

Presented at the IEEE Conference on real time computer applications in nuclear,
particle and plasma physics, Williamsburg, Virginia, May 16-19, 1989

SIMULATION OF A MACRO-PIPELINED MULTI-CPU EVENT PROCESSOR FOR USE IN FASTBUS

M. F. Letheren, A. Marchioro
*CERN European Organisation for Nuclear Research
1211 Geneva 23, Switzerland*

F. Slorach
*Rutherford Appleton Laboratory
Chilton, Didcot OX11 0QX England*

Abstract

The aim of this paper is to demonstrate how commercially available hardware description languages, which are normally used for hardware design verification at low to medium levels of modeling abstraction, can be used to optimize the design of complex real-time systems exhibiting concurrency. We model a high performance, loosely-coupled, multi-processor FASTBUS master intended for real-time event processing applications. The performance of such a system depends on a large number of parameters that interact in a complex way, so that it becomes very difficult to make reliable tradeoffs in the design of the system architecture using an analytical approach. Given a high level behavioural hardware description language and its simulator it is relatively easy to develop an executable model that provides a test bench for optimization of the architecture.

Introduction

Large multi-crate FASTBUS data acquisition systems are typically built in a tree structure. Data originate at the base of the tree and flow upwards through several levels of embedded processors that perform data reduction, assembly of events, or event rejection, until finally the data rate at the root of the tree can be handled by a data acquisition computer. Multiple events can be pipelined through the various processing steps carried out at different levels of the tree.

In order to speed throughput one can use hardwired processors or special micro-programmable processors at the lower levels, assigning a few channels to each. At the higher levels, processors tend to require access to the complete set of an event's data and algorithms tend to be more time consuming. At this level the use of multiple, high performance, commercial Reduced Instruction Set Computers (RISCs) can boost performance by processing events in a pipeline. Different independent tasks are executed in different stages of this macro-pipeline, and processors are dedicated to each stage of the pipeline; for instance one processor controls FASTBUS readout, another does data re-formatting, and another does peak finding in calorimeter data. This approach differs from the tightly coupled multi-processor architectures where all processors work on the same event at the same time.

The Architecture of the multi-CPU event processor

Many possible macro-pipelined, multi-cpu architectures are possible. Figure 1 shows a very simple architecture that uses a minimum of buses, buffer memories, and control logic etc. The system is built around single chip, high performance, commercially available, 32-bit SPARC™ RISC processors [1] running at clock frequencies of 20 MHz or above [Note 1] and capable of delivering effective performance in excess of 12 million instructions per second (MIPS).

We model a 4 processor system in which a shared (global) data memory is used to buffer events while they are being processed through the various stages of the macro-pipeline. The global data memory size is of the order 1 MB and it has a cycle time

Fig. 1 P_x Processor, IM Instruction memory, FBM Fastbus Master interface, DC Data Cache, CC Cache Controller, FBS Fastbus Slave interface, GDM shared Global Data Memory

of 100 ns. The global data memory is also used to store program data. Each processor has its own fast instruction memory which is accessed over the 32-bit wide synchronous processor bus (PBUS). A high speed virtual cache for data and its controller sit directly on each processor's PBUS.

A high performance, synchronous memory bus (MBUS) provides a 64-bit wide data path between global data memory and each processor's PBUS. The MBUS usually transfers data in blocks, each block transfer requiring one bus arbitration cycle, one address cycle, and a number of data cycles. When the MBUS is clocked at the same speed as the processors (20 MHz) and the global memory can cycle at 100 ns, a 64-bit data word can be transferred every second clock cycle within the block transfer. The memory subsystem contains logic to increment the address registers after each memory cycle.

Each cache controller chip manages the data cache and provides the interface between the MBUS protocol and the PBUS protocol. It also maps addresses produced by the processor into physical addresses in the global data memory (allowing shared pages for inter-processor communication), and provides memory protection. Memory mapping functions are set up under the control of a real-time operating system running on the processor.

FASTBUS data are read into the global memory under the control of processor P0, which has a FASTBUS master interface implemented as a semi-custom ASIC co-processor. FASTBUS data are not cached in the data cache of P0 during readout. After processing, the event data are read out of global memory by a FASTBUS master operating from a higher level in the data acquisition tree. This function is supported by a FASTBUS slave interface ASIC that interfaces directly to the MBUS.

Cache memory operation

On each processor the data cache is a 64 KB memory built from static RAMs with 20 ns access time. The cache controller implements several modes of operation that allow minimization of the cache miss ratio and the penalty incurred when a miss does occur. The optimal cache operation mode will depend on the application and system architecture in which the cache is embedded.

The cache relies on the principle of locality of information, i.e. data near the currently referenced memory address are likely to be referenced in the near future. Detailed description of cache memory architecture and performance measurements can be found in reference [2]. The quantum of data cached is a "line", which is a block of data from contiguous addresses in global memory, typically of length 32 bytes. When operated in the "demand fetch" mode, the cache controller fetches a line into cache whenever a read miss occurs. In this mode the cache controller does not prefetch other lines.

Write operations could be handled either by the "copy-back" or the "write-through" strategy. In copy-back mode the processor normally writes data into the cache. Whenever it is necessary to empty a cache line to global memory we assume that the

data are transferred into "write back" registers in the cache controller and the processor continues execution while global memory is updated by the controller as soon as it can get MBUS mastership. On the other hand, in the write-through mode every write cycle updates the cache and global memory via the write back registers and the MBUS.

In our simple macro-pipelined application the traffic on the MBUS impacts the performance of all cpus by slowing down caching between global memory and the processor data caches. This traffic will depend on many factors, including the cache size, the cache line-length, the choice of copy-back versus write-through policy, whether cache prefetching is implemented, the size of the physics data structures, and the algorithms, programming language and programming style used by the applications programmer. It is therefore very hard to estimate system performance without an executable model that emulates the operation of the buses, caches and algorithms running in the processor. The model must predict the effect of these parameters on system performance when running with physics code and data. In order to do so, the concurrency and timing of all aspects of the system operations must be modeled realistically.

Modeling the system using a hardware description language

Rather than attempt to model the concurrency and timing of the system in a procedural language such as C, Pascal, or Fortran we used the high-level, behavioural modeling capability of a hardware description language (HDL). Because such HDLs, together with their simulators, are tailored for modeling the timing and concurrency of hardware systems, it is possible to generate models with comparatively little effort. The disadvantage is that the model execution speed will be slower than a model coded in C for example, so that one is forced to trade off model detail for speed. We chose to use the Verilog[®] HDL and its interactive simulator [3] because, in addition to supporting description of timing and concurrency, it contains primitives that facilitate the stochastic analysis of systems.

Choice of the level of modeling abstraction

Although the HDL contains all the semantics needed for accurate description of the behaviour of hardware, a heavy penalty is paid in the overhead needed by the simulator to simulate the execution of multiple concurrent activities and to schedule events with their associated delays. For example, we have developed a detailed model of the SPARC RISC integer unit at the register transfer level (description of a network of components such as registers, ALUs, shifters, PLAs, etc.). This runs compiled object code at a speed of a few machine instructions per second on a Sun-4/260 workstation (more than six orders of magnitude slower than the real hardware); a speed that is clearly inadequate for modeling our multi-processor system running physics code.

In our multi-processor system the main point is to study how performance depends on the gross features of the architecture; i.e. the connectivity and bandwidth of buses, cache memory system configuration, data buffering between modules or within the readout, etc. The randomness of event arrival times and the

spread of event data size suggests that a stochastic analysis (MonteCarlo) approach is appropriate. It therefore seems adequate to model the processors as entities that, while they are processing event data, generate cache read and write misses randomly in time at some parameterised mean rate [Note 2].

By eliminating the detailed internal operation of the processors from the model we achieve acceptable execution speeds. The accuracy of the model then depends on obtaining reliable estimates of the miss rates for physics code typical of that to be run on the processors, and for various possible configurations of the cache memory sub-systems. We extract the cache miss rate parameters for our model by running code on a more efficient emulation of the RISC instruction set (written in C and without detailed modeling of timing and concurrency).

The cache controllers are modeled as units that generate block transfers to the global memory over the MBUS whenever their associated processor generates a cache miss. Copy-back and write-through modes are selectable. The write-through to global memory assumes that the processor continues execution once the write-back registers have been loaded and that the global memory update then goes on in the background. The MBUS arbitration is described accurately so that it reproduces the degradation of the wait time necessary to obtain bus mastership as traffic increases.

Primitives for stochastic analysis of systems

The Verilog HDL and simulator provide primitives facilitating stochastic analysis of systems. Random number generators are available to produce values following several well known statistical distributions (uniform, normal, exponential, Poisson, chi-squared, and others). A set of tasks for management and measurement of queues supports the modeling of communication between processes at a high level of abstraction.

The multi-processor system model contains a module that represents the experiment and the FASTBUS data acquisition system that delivers readout data. This generates events randomly in time at a parameterised nominal rate, and with an exponential distribution in event size between a lower and an upper limit. Any arbitrary depth of buffering is modeled in the front-end detectors allowing the detector to be "live" while readout is in progress. Deadtime losses are recorded when processing times are so long that the global memory becomes full, causing further event triggers to be inhibited. A FIFO buffer is modeled in the FASTBUS coprocessor interface in order to smooth the FASTBUS and MBUS traffic. A slave readout process is also modeled in order to remove processed events from the global memory.

The scheduling of events through the macro-pipeline is modeled using the simulator's queuing primitives; each processor takes events from a queue and places the processed events in another queue. By changing the pointers to each processor's event input and output queues one can easily reconfigure the scheduling of events through the processors, modeling, for example, a pure macro-pipeline, a load-balanced macro-pipeline (using different numbers of processors at each stage), or the

commonly used processor "farm" where each processor runs the complete event analysis code and there is no pipelining of tasks.

The human interface

The power of the simulator's primitives for graphical display of results should not be underestimated. A simple statement analogous to the C-language *printf* statement is sufficient to set up a window displaying the contents of registers, or parameter values. The simulator updates this display automatically (i.e. without explicit user action) whenever a displayed variable changes value. Similar high level statements allow the user to set up windows that display automatically updated bar-charts. Using these features one can instrument the model and produce dynamic graphical display of the measurements (e.g. percentage utilisation of various system resources, queuing times, etc.). The conventional waveform display is also available if needed (we mainly used this to debug the details of the model, rather than to visualize simulation results).

Our basic instrumented model consists of approximately 1900 lines of code and was developed with 4 man-weeks of effort. Run times depend strongly on event size, the assumed profile of processor compute time per event, and the cache miss rates, but typically lie between a few seconds and a few 10's of seconds per event. The use of the modular description style encouraged by the HDL makes changes in order to explore variants on the architecture quite straight forward. For example, when it seemed that the MBUS was a bottleneck, it required 2 hours effort to cut the MBUS into two segments as shown in Figure 2, where each seg-

Fig.2 Dual bus multi-processor system with pseudo-dual ported memory organized in eight non-interleaved banks.

ment accesses a global memory arranged in 8 banks, each of which is accessed via a single port and an arbitration circuit.

Estimation of data cache performance

Our model of the RISC processor generates data cache misses pseudo-randomly at a rate characteristic of the code and data structures used for event analysis. We use an architectural simulator of the RISC instruction set to produce program execution traces, which are then analysed by a cache memory simulator to derive cache miss rates for a specific cache memory configuration.

The SPARCSim instruction level simulator

We have used the SPARCSim architectural simulator [4] to produce execution traces for the following C language programs:

- (1) An algorithm that builds a hash-table to look up pointers to the many data blocklets that constitute a complete event, and that can arrive in any order due to the asynchronous pipelining of event data through the various branches of the FASTBUS tree.
- (2) A peak finding algorithm for a 256 X 256 array [5].
- (3) A general purpose histogramming package [6] used to fill 50 one-dimensional histograms of 100 bins with a total of 1000 integer entries.

SPARCSim includes emulation of trap handlers and run time support such as that provided by the SUNOSTM operating system, and it can be configured to emulate any user defined operating system kernel. We have not simulated the overhead associated with operating system functions. Because SPARCSim does not model concurrency and detailed timing it is able to emulate code at approximately 13,000 instructions per second on a Sun-4/260 workstation.

The Dinero cache performance analysis tool

We used the DineroIII trace-driven cache simulator [7] to derive data cache performance figures from the execution traces produced by SPARCSim. A trace extraction utility supplied with SPARCSim was used to build the memory reference trace file needed to drive Dinero. This contains a record for each memory access, specifying the memory access type (one of "read data", "write data", "instruction fetch", "unknown access type", or "cache flush") and the memory address. DineroIII allows simulation with many possible cache configurations; for example one can select line length, cache size, separate or combined instruction and data cache, cache associativity, replacement policy, fetch policy, write policy, etc.

Table 1 shows results obtained with Dinero for the 3 program traces described above using a 64 KB data cache operating with a line length of 32 bytes, and with no prefetching. The figures show that cache performance is strongly dependent on the algorithms used and the size of the data structures relative to the cache size. Therefore careful optimization of algorithms and coding

using these tools can produce significant performance improvements.

Table 1: Data cache trace analysis results

Trace	Source lines	Trace length (cycles)	Write policy	Cache misses (% of cycles)	
				read	write
Hash table	170	28000	write-thru	0.5	5.1
			copy-back	0.5	3.5
Peak find	50	13600000	write-thru	3.4	1.8
			copy-back	3.4	11.1
Histogram	1800	795 000	write-thru	0.1	5.3
			copy-back	0.1	0.8

Typical measurements and comparison of two architectures

Figure 3 shows the effective throughput of the multi-processor system as a function of the number of processors in the system for different values of the cache miss rate (where 50% are assumed to be write misses). The effective throughput measured with the simulator is normalised to the throughput of a single processor running with 100% data cache hits. These figures are derived for the case where no processor ever idles waiting for a new event to arrive (a perfectly load-balanced macro-pipeline kept fully busy), so that the MBUS is driven as hard as possible.

Because performance degrades with the saturation of the MBUS we looked at a modified system architecture, shown in Figure 2, that splits the MBUS into two independent segments. The global memory address space is divided into 8 non-inter-

Fig.3 Effective throughput of the multi-processor system for two different architectures and data cache miss ratios. (A) 4% miss, 2 buses; (B) 4% miss, 1 bus; (C) 8% miss, 2 buses; (D) 8% miss, 1 bus; (I) Ideal curve, for no cache misses.

leaved banks. The two MBUS segments must arbitrate for access to each memory bank. The arbitration for bank access introduces an additional bus cycle delay in each block transfer, but the two segments can access different banks in parallel. Figure 3 also shows how this modification improves the multi-processor system efficiency.

Fig.4 Deadtime and event collection efficiency as functions of event rate for a 4 processor macro-pipelined multi-processor and a typical set of model parameter values.

Figure 4 shows how the deadtime introduced by the event processing macro-pipeline depends on the event rate for a typical set of parameters characterizing a four processor system with a 1 MB global data memory. These measurements were made with a sample of events of between 50 and 200KB size, with processing times modeled as linear functions of event size (the slowest processor taking 11 ms/event without latency due to cache misses), and with a mix of data cache miss ratios between 3 and 7 percent.

Conclusions

We have described a methodology that uses behavioural hardware description languages to create a test bench for optimization of the architectural design of complex real-time systems containing many concurrently operating sub-systems, and where system performance may depend on a large number of design parameters that interact in a complex way making analysis by analytical methods impractical. System performance is then studied using stochastic analysis techniques, much as physicists use MonteCarlo techniques to optimize and understand the performance of their detectors. Model development times using an HDL and simulator are faster than using a normal procedural language, but simulation speed is slower. We believe that, with an appropriate choice of the level of modeling abstraction, this methodology can be used to verify and optimize the architectures of a class of parallel real-time systems similar to the one presented in this paper. Another example could be multiple FASTBUS masters operating in a FASTBUS data acquisition system.

Notes

(1) The RISC architecture used is the SPARC [1], available from several sources in a number of implementations of different performance and packaging. Currently 33 MHz versions are available, but here we use a conservatively slow clock speed of 20 MHz to reflect the likely difficulty of constructing a large synchronous system containing several processors running at higher frequencies. The model described in this paper could be applied, with minor changes, to similar architectures based on RISC processors from other sources.

(2) Although cache misses are not completely randomly distributed in reality, an 8 word cache write through buffer smoothes the effect of write bursts (analysis with the architectural simulator has shown that data cache read or write bursts longer than 8 words are very rare. We believe that, on the macroscopic scale, cache misses can be treated as occurring randomly without serious inaccuracy

Acknowledgements

We thank Prof Antonio Zichichi, Horst Wenninger and David Williams for encouraging us in this work. We are grateful to Ed Gronke of Sun Microsystems for assistance with SPARCSim, Mark Hill of the University of Wisconsin for providing the DineoroIII cache simulator, and our CERN colleagues Rudy Bock, Werner Krischer, Sigurd Lone, and Andre Lacourt, for providing the code used to generate the simulation traces.

SPARC, SUNOS and Sun-4 are trademarks of Sun Microsystems, Inc. Verilog is a trademark of Gateway Design Automation Corp.

References

- [1] Sun Microsystems Inc., The SPARC Architecture Manual, Rev A October, 1987.
- [2] A. J. Smith, "Cache Memories", Computing Surveys, 14-3, September 1982, pp. 473-530.
- [3] Gateway Design Automation Corp., Verilog Hardware Description Language and Simulator Reference Manual, Version 1.2, 1988.
- [4] Sun Microsystems Inc., SPARCSim User's Guide, Rev. A, December 1988.
- [5] R. Bock, W. Krischer, S. Lone, "Benchmarking computer architectures for HEP real time algorithms", CERN-LAA RT/88-08, November 1988.
- [6] A subset of the HBOOK package (CERN Program Library Y250).
- [7] DineoroIII, a cache analysis tool kindly made available to us by M. D. Hill, Computer Science Dept., Univ. of Wisconsin, Madison WI 53706.