Modular Software Performance Monitoring

Daniele Francesco Kruse – CERN – PH / SFT

Karol Kruzelecki – CERN – PH / LBC
Summary

1. Motivation and Goal

2. Introduction
   • Performance Monitoring
   • Performance Counters

3. Challenge
   • Monolithic vs. Modular Monitoring
   • Modularity in CMSSW, Gaudi and Geant4

4. Solution
   • Analysis Overview
   • What do we monitor?

5. Simple example of successful usage

6. Conclusions
Motivation and Goal

- **HEP** software is huge and complex and is developed by a multitude of programmers often unaware of performance issues.

- The software produced is suboptimal in terms of efficiency and speed.

- Unfortunately CPU speed is not likely to be increased in the near future as we were used to in the past.

- The **goal** is then to find an effective method to improve SW through **monitoring and optimization**.

- Better performance (more **throughput**) means **savings** both in hardware and power needed.
Performance Monitoring

**DEF**: The action of collecting information related to how an application or system performs

**HOW**: Obtaining micro-architectural level information from hardware performance counters

**WHY**: To identify bottlenecks, and possibly remove them in order to improve application performance
Performance Counters

• All recent processor architectures include a processor–specific PMU

• The Performance Monitoring Unit contains several performance counters

• Performance counters are able to count micro-architectural events from many hardware sources (cpu pipeline, caches, bus, etc…)

• We focus on the two main Intel® cpu families currently on the market: Core and Nehalem

• Nehalem processors feature 4 programmable counters while Core processors have 2 programmable counters
Monolithic vs. Modular Monitoring

For each CPU performance event that we monitor...

1 RUN

Monolithic “Black Box”

1 set of results

Module A

Module B

Module C

Event N

Event 3

Event 2

Event 1

Res. A1 Res. A2 Res. A3 ... Res. AN

AVG.

Res. A

Res. B1 Res. B2 Res. B3 ... Res. BN

AVG.

Res. B

Res. C1 Res. C2 Res. C3 ... Res. CN

AVG.

Res. C

set of results for A

set of results for B

set of results for C

1 RUN

set of results

RUN
Monolithic vs. Modular Monitoring

- When we face large and complex software monolithic analysis becomes less useful

- “Traditional” monitoring tools (using performance counters) are monolithic. Examples: PTU and pfmon

- Even sampling over symbols (functions) is not enough for code division. Solution: modular monitoring!

- Code instrumentation (minimal in HEP software) and ad-hoc interface to the monitoring tool needed

- Advantage: narrowing down the possible location of performance problems leads to easier optimization
Modularity in HEP SW

- **CMSSW** code is organized into *modules* that are sequentially executed during each event processed, and it provides *hooks* to execute user-defined actions at the beginning and at the end of modules.

- Hooks is what we use to *start* and *stop* the *monitoring process* and to collect results for each module.

- More on **CMSSW** performance in Matti Kortelainen’s talk.

- **Gaudi** provides a similar mechanism to instrument its code (modules are called *algorithms*).

- **Geant4** is handled differently: binning into triples $\langle$particle type, energy range, physical volume$\rangle$. 
Analysis flow-graph

Start

Analysis Configuration

Performance Data Taking

Program Run

Performance Data Output

CMSSW or Gaudi or Geant4

Performance Data Analysis

Browseable HTML results

End
What and why do we monitor?

- Total Cycles (Application total execution time)
  - Issuing μops
    - Retiring μops (useful work)
    - Not retiring μops (useless work)
  - Not Issuing μops
    - Stalled (no work)

Subcategories:
- Load Stalls
- FP Exceptions
- Divs & Sqrts
- Ifetch misses
- Branches
Example of usage – LHCb (Gaudi)

• Tested on *Brunel v37r7*
  
  **GaudiRun** <options>

  **GaudiProfiler** <options>

  No code instrumentation needed

• **GaudiProfiler** – python script handling sequential run of application for all the necessary counters and postprocessing

<table>
<thead>
<tr>
<th>MODULE NAME</th>
<th>Total Cycles</th>
<th>Instructions Retired</th>
<th>CPI</th>
<th>Margin</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FitBest</td>
<td>88172186.85</td>
<td>113337618.59</td>
<td>0.78</td>
<td>11.94</td>
<td>2.28</td>
</tr>
<tr>
<td>CreateOfflinePhotons</td>
<td>62577023.13</td>
<td>53160705.52</td>
<td>1.18</td>
<td>9.84</td>
<td>1.83</td>
</tr>
<tr>
<td>FitReadForMatch</td>
<td>38233322.96</td>
<td>52203155.51</td>
<td>0.73</td>
<td>5.03</td>
<td>0.74</td>
</tr>
<tr>
<td>MuonIDAlg</td>
<td>36209823.66</td>
<td>36911424.03</td>
<td>0.90</td>
<td>5.39</td>
<td>0.63</td>
</tr>
<tr>
<td>RichofflineGPIDLL1t0</td>
<td>29723192.21</td>
<td>32871302.13</td>
<td>0.90</td>
<td>9.29</td>
<td>0.98</td>
</tr>
<tr>
<td>CreateOfflineTracks</td>
<td>19249568.95</td>
<td>20883977.44</td>
<td>0.92</td>
<td>2.80</td>
<td>0.35</td>
</tr>
<tr>
<td>FitVelo</td>
<td>17809787.32</td>
<td>19615917.09</td>
<td>0.91</td>
<td>2.58</td>
<td>0.31</td>
</tr>
<tr>
<td>RichofflineGPIDLL1t1</td>
<td>17759552.39</td>
<td>18178152.35</td>
<td>0.98</td>
<td>2.64</td>
<td>0.88</td>
</tr>
<tr>
<td>TkaSeed</td>
<td>15595472.48</td>
<td>16022706.13</td>
<td>0.97</td>
<td>2.31</td>
<td>0.33</td>
</tr>
<tr>
<td>PatVeloTT</td>
<td>11575267.16</td>
<td>13407311.72</td>
<td>0.86</td>
<td>1.64</td>
<td>0.22</td>
</tr>
</tbody>
</table>
Choice of an algorithm

- **Objective:** to reduce the number of *Total cycles* (execution time) of one *algorithm*
- As a simple example we choose to focus on reducing the cycles in which instructions were retired
- **How:** This is done by reducing the number of *Instructions Retired* – a very stable and reliable counter

<table>
<thead>
<tr>
<th>MODULE NAME</th>
<th>Total Cycles ▲</th>
<th>Instructions Retired</th>
<th>CPI</th>
<th>iMargin</th>
<th>iFactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FitBest</td>
<td>88172186.85</td>
<td>11339618.59</td>
<td>0.78</td>
<td>11.94</td>
<td>2.28</td>
</tr>
<tr>
<td><strong>CreateOfflinePhotons</strong></td>
<td><strong>62577023.13</strong></td>
<td><strong>53160705.52</strong></td>
<td><strong>1.18</strong></td>
<td><strong>9.84</strong></td>
<td><strong>1.83</strong></td>
</tr>
<tr>
<td>FitSeedForMatch</td>
<td>38233332.96</td>
<td>52203155.51</td>
<td>0.73</td>
<td>5.03</td>
<td>0.74</td>
</tr>
<tr>
<td>MuonIDAlg</td>
<td>36209823.66</td>
<td>36911424.03</td>
<td>0.98</td>
<td>5.39</td>
<td>0.63</td>
</tr>
</tbody>
</table>
Symbols – Looking deeper in

- After choosing *Algorithm*, in the *detailed symbol view* → .cpp file and function

- *Inline*d functions are not shown, they are counted in the “parent” functions

<table>
<thead>
<tr>
<th>Samples</th>
<th>Percentage</th>
<th>Symbol Name</th>
<th>Library Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>6911</td>
<td>13.435854%</td>
<td>solveQuarticEq</td>
<td>libRichRecPhotonTools.so</td>
</tr>
<tr>
<td>2042</td>
<td>3.969905%</td>
<td>reconstructPhoton</td>
<td>libRichRecBase.so</td>
</tr>
<tr>
<td>1977</td>
<td>3.843537%</td>
<td>photonPossible</td>
<td>libRichRecPhotonTools.so</td>
</tr>
<tr>
<td>1867</td>
<td>3.629683%</td>
<td>reconstructPhoton</td>
<td>libRichRecPhotonTools.so</td>
</tr>
</tbody>
</table>
Detailed profiling of one function

- We modify the body of function adding `start()` and `stop()` commands for profiler
- Results is shown after the run of application is over

```cpp
#include "GaudiProfiling/PfmCodeAnalyser.h"

PfmCodeAnalyser::Instance("INSTRUCTIONS RETIRED").start();

Event: INSTRUCTIONS RETIRED
Total count:1697360246
Number of counts:1548592
Average count:1096.066779
Overhead removed:42

PfmCodeAnalyser::Instance().stop();
```
Improving small parts of code

- Optimization procedure loop:
  1. modify code
  2. compile
  3. profile

- Compare average count of Instructions Retired

-6.5%
**Re-run after changes**

- Even small changes are visible in *Instructions Retired*.
- *Total Cycles* decreased – it is faster.

<table>
<thead>
<tr>
<th>MODULE NAME</th>
<th>Total Cycles</th>
<th>Instructions Retired</th>
<th>CPI</th>
<th>iMargin</th>
<th>iFactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FitBest</td>
<td>88172186.85</td>
<td>113337618.59</td>
<td>0.78</td>
<td>11.94</td>
<td>2.28</td>
</tr>
<tr>
<td>CreateOfflinePhotons</td>
<td>62577923.13</td>
<td>53160705.52</td>
<td>1.18</td>
<td>9.84</td>
<td>1.83</td>
</tr>
<tr>
<td>FitSeedForMatch</td>
<td>38233332.96</td>
<td>52203155.51</td>
<td>0.73</td>
<td>5.03</td>
<td>0.74</td>
</tr>
<tr>
<td>MuonIDAAlg</td>
<td>36209823.66</td>
<td>36911424.03</td>
<td>0.98</td>
<td>5.39</td>
<td>0.63</td>
</tr>
</tbody>
</table>

- ~6.5% improvement in one function gave ~2% improvement in the *algorithm*.
Conclusions

- We implemented a modular ad-hoc **performance counters-based monitoring tool** for three major HEP frameworks: *CMSSW*, *Gaudi* and *Geant4*

- This tool is supposed to help developers optimizing existing code to **improve its performance** without the need for code instrumentation

- The tool has been successfully used to optimize code in *Gaudi* and has shown the potential to be used for other applications as well

- *GaudiProfiling* package will be available in the next release of *Gaudi*
Thank you, Questions ?
backup slides
### BACKUP: The 4-way Performance Monitoring

<table>
<thead>
<tr>
<th>Counting</th>
<th>Overall (pfmon)</th>
<th>Modular</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Overall Analysis</td>
<td>3. Module Level Analysis</td>
</tr>
<tr>
<td>Sampling</td>
<td>2. Symbol Level Analysis</td>
<td>4. Modular Symbol Level Analysis</td>
</tr>
</tbody>
</table>
BACKUP: *Core* and *Nehalem* PMUs - Overview

**Intel Core Microarchitecture PMU**

- 3 fixed counters
  (INSTRUCTIONS RETIRED, UNHALTED_CORE_CYCLES, UNHALTED_REFERENCE_CYCLES)
- 2 programmable counters

**Intel Nehalem Microarchitecture PMU**

- 3 fixed *core*-counters
  (INSTRUCTIONS RETIRED, UNHALTED_CORE_CYCLES, UNHALTED_REFERENCE_CYCLES)
- 4 programmable *core*-counters
- 1 fixed *uncore*-counter (UNCORE_CLOCK_CYCLES)
- 8 programmable *uncore*-counters
A generic API to access the PMU (libpfm)
Developed by Stéphane Eranian
Portable across all new processor micro-architectures
Supports system-wide and per-thread monitoring
Supports counting and sampling

**BACKUP: Perfmon2**

- Pfmon
- Other libpfm-based Apps
- User space
- Generic Perfmon
- Architectural Perfmon
- libpfm
- Linux Kernel
- PMU
- CPU Hardware
**Nehalem**: Overview of the architecture

### Core 0
- TLBs
- L1I Cache
- L1D Cache
- L2 Cache

### Core 1
- DTLB0 & ITLB (1st Level), STLB (unified 2nd Level)
- writeback

### Core 2
- writeback

### Core 3
- unified, writeback, not-inclusive

### Level 3 Cache
- shared, writeback (lazy write) and inclusive (contains L1 & L2 of each core)

### Integrated Memory Controller
- Link to local memory (DDR3)

### Quick Path Interconnect
- Link to I/O hub
  (& to other processors, if present)
We are mainly interested in **UOPS_EXECUTED** (dispatched) and **UOPS RETIRED** (the useful ones).

Mispredicted **UOPS_ISSUED** may be eliminated before being executed.
BACKUP: Cycle Accounting Analysis

Total Cycles (Application total execution time)

Issuing μops

Retiring μops (useful work)

Not retiring μops (useless work)

Not Issuing μops

Stalled (no work)

L2 miss

L2 hit

L1 TLB miss

Store-Fwd

LCP
BACKUP: New analysis methodology for Nehalem

**BASIC STATS:** Total Cycles, Instructions Retired, CPI;

**IMPROVEMENT OPPORTUNITY:** iMargin, iFactor;

**BASIC STALL STATS:** Stalled Cycles, % of Total Cycles, Total Counted Stalled Cycles;

**INSTRUCTION USEFUL INFO:** Instruction Starvation, # of Instructions per Call;

**FLOATING POINT EXCEPTIONS:** % of Total Cycles spent handling FP exceptions;

**LOAD OPS STALLS:** L2 Hit, L3 Unshared Hit, L2 Other Core Hit, L2 Other Core Hit Modified, L3 Miss -> Local DRAM Hit, L3 Miss -> Remote DRAM Hit, L3 Miss -> Remote Cache Hit;

**DTLB MISSES:** L1 DTLB Miss Impact, L1 DTLB Miss % of Load Stalls;

**DIVISION & SQUAREROOT STALLS:** Cycles spent during DIV & SQRT Ops;

**L2 IFETCH MISSES:** Total L2 IFETCH misses, IFETCHes served by Local DRAM, IFETCHes served by L3 (Modified), IFETCHes served by L3 (Clean Snoop), IFETCHes served by Remote L2, IFETCHes served by Remote DRAM, IFETCHes served by L3 (No Snoop);

**BRANCHES, CALLS & RETS:** Total Branch Instructions Executed, % of Mispredicted Branches, Direct Near Calls, Indirect Near Calls, Indirect Near Non-Calls, All Near Calls, All Non Calls, All Returns, Conditionals;

**ITLB MISSES:** L1 ITLB Miss Impact, ITLB Miss Rate;

**INSTRUCTION STATS:** Branches, Loads, Stores, Other, Packed UOPS;
**BACKUP: PfmCodeAnalyser, fast code monitoring**

- Unreasonable (and useless) to run a complete analysis for every change in code

- Often interested in only small part of code and in one single event

- Solution: a fast, precise and light “singleton” class called *PfmCodeAnalyser*

- How to use it:

  ```
  #include<PfmCodeAnalyser.h>

  PfmCodeAnalyser::Instance("INSTRUCTIONS RETIRED").start();

  //code to monitor

  PfmCodeAnalyser::Instance().stop();
  ```
PfmCodeAnalyser::Instance("INSTRUCTIONS_RETIRED", 0, 0,
    "UNHALTED_CORE_CYCLES", 0, 0,
    "ARITH:CYCLES_DIV_BUSY", 0, 0,
    "UOPS_RETIRED:ANY", 0, 0).start();

Event: INSTRUCTIONS_RETIRED
    Total count:105000018525
Number of counts:10
    Average count:10500001852.5

Event: UNHALTED_CORE_CYCLES
    Total count:56009070544
Number of counts:10
    Average count:5600907054.4

Event: ARITH:CYCLES_DIV_BUSY
    Total count:28000202972
Number of counts:10
    Average count:2800020297.2

Event: UOPS_RETIRED:ANY
    Total count:138003585913
Number of counts:10
    Average count:13800358591.3
BACKUP: *What can we do with counters?*

**Question:**

*Is all this useful?*

**Answer:**

*We don’t know, but we shall see*

- Lack of papers and literature about the subject

- An empirical study is underway to find out:
  1. A relationship between counter results and coding practices
  2. A practical procedure to use counter results to optimize a program

- A procedure has already been developed and will be tested

- The trial study will be conducted on Gaudi together with *Karol Krużelecki* (PH-LBC group)
BACKUP: The 3-step optimization procedure

- We start from counter results and choose one algorithm to work on using the *Improvement Margin* and the *iFactor*.

- We then apply the following procedure:

1. Change to a more efficient algorithm and vectorize it
2. Remove stall sources (*L1 & L2 misses, store-fwd, etc.*.)
3. Remove misprediction sources (*branches, calls, etc.*.)
BACKUP: Overall Analysis

- Uses Pfmon and it is based on the *Cycle Accounting Analysis*

- Good for showing **overall performance** and for checking improvements

- Good for identifying general software problems

- Good for comparing different versions of the code

- *NOT* enough for
  - finding inefficient parts of the software
  - finding bad programming practices
BACKUP: Overall Analysis

TOTAL CYCLES: 1408291621561

Cycle composition at issue port
- Cycles not issuing uops: 33.819%
- Cycles issuing uops: 66.181%

Cycle composition of OOO execution
- Cycles not retiring uops: 1.664%
- Cycles retiring uops: 61.389%
- Cycles stalled: 36.947%

Stalls composition
- L2 miss impact: 7.783%
- L2 hit impact: 13.703%
- L1 dtlb miss impact: 3.233%
- LCP stalls impact: 0.261%
- Store fwld stalls impact: 75.020%

Store-forward stalls composition
- Loads blocked by unknown address store: 74.672%
- Loads overlapped with stores: 22.769%
- Loads spanning across cache line: 2.560%
BACKUP: Symbol Level Analysis

- Uses *sampling* capabilities of pfmon

- Good for identifying general **bad programming practices**

- Can identify problems of functions which are frequently used

- Shows **functions** that use most of the execution cycles and functions that spend a lot of time doing nothing (stalling)

- **NOT** good for finding specific problems in the code
## BACKUP: Symbol Level Analysis

### Total Cycles

<table>
<thead>
<tr>
<th>counts</th>
<th>%self</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>54894</td>
<td>3.79%</td>
<td>_int_malloc</td>
</tr>
<tr>
<td>50972</td>
<td>3.52%</td>
<td>_GI_libc_malloc</td>
</tr>
<tr>
<td>41321</td>
<td>2.85%</td>
<td>__cfree</td>
</tr>
<tr>
<td>36294</td>
<td>2.51%</td>
<td>ROOT::Math::SMatrix::operator=</td>
</tr>
<tr>
<td>31100</td>
<td>2.15%</td>
<td>_ieee754_exp</td>
</tr>
<tr>
<td>25636</td>
<td>1.77%</td>
<td>ROOT::Math::SMatrix::operator=</td>
</tr>
<tr>
<td>24833</td>
<td>1.72%</td>
<td>do_lookup_x</td>
</tr>
<tr>
<td>23206</td>
<td>1.60%</td>
<td>ROOT::Math::SMatrix::operator=</td>
</tr>
<tr>
<td>22970</td>
<td>1.59%</td>
<td>_ieee754_log</td>
</tr>
<tr>
<td>21741</td>
<td>1.50%</td>
<td>__atan2</td>
</tr>
<tr>
<td>20467</td>
<td>1.41%</td>
<td>ROOT::Math::SMatrix::operator=</td>
</tr>
<tr>
<td>19922</td>
<td>1.38%</td>
<td>_int_free</td>
</tr>
<tr>
<td>18354</td>
<td>1.27%</td>
<td>G_defined_typename</td>
</tr>
<tr>
<td>16026</td>
<td>1.11%</td>
<td>strcmp</td>
</tr>
<tr>
<td>15979</td>
<td>1.10%</td>
<td>TList::FindLink</td>
</tr>
<tr>
<td>14601</td>
<td>1.01%</td>
<td>G_defined_tagname</td>
</tr>
</tbody>
</table>

### Stalled Cycles

<table>
<thead>
<tr>
<th>counts</th>
<th>%self</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>24955</td>
<td>5.09%</td>
<td>_int_malloc</td>
</tr>
<tr>
<td>19797</td>
<td>4.04%</td>
<td>do_lookup_x</td>
</tr>
<tr>
<td>19084</td>
<td>3.89%</td>
<td>_GI_libc_malloc</td>
</tr>
<tr>
<td>14282</td>
<td>2.91%</td>
<td>_ieee754_exp</td>
</tr>
<tr>
<td>13564</td>
<td>2.77%</td>
<td>strcmp</td>
</tr>
<tr>
<td>13065</td>
<td>2.66%</td>
<td>__cfree</td>
</tr>
<tr>
<td>9927</td>
<td>2.02%</td>
<td>__atan2</td>
</tr>
<tr>
<td>8998</td>
<td>1.83%</td>
<td>_ieee754_log</td>
</tr>
<tr>
<td>7666</td>
<td>1.56%</td>
<td>TList::FindLink</td>
</tr>
<tr>
<td>7575</td>
<td>1.54%</td>
<td>_int_free</td>
</tr>
<tr>
<td>5392</td>
<td>1.10%</td>
<td>std::basic_string::find</td>
</tr>
<tr>
<td>4911</td>
<td>1.00%</td>
<td>computeFullJacobian</td>
</tr>
<tr>
<td>4410</td>
<td>0.90%</td>
<td>malloc_consolidate</td>
</tr>
<tr>
<td>4285</td>
<td>0.87%</td>
<td>operator new</td>
</tr>
<tr>
<td>4104</td>
<td>0.84%</td>
<td>ROOT::Math::SMatrix::operator=</td>
</tr>
<tr>
<td>3949</td>
<td>0.81%</td>
<td>33.84% makeAtomStep</td>
</tr>
</tbody>
</table>
BACKUP: Module Level Analysis

- Uses the Perfmon2 interface (libpfm) directly

- Analyses each CMSSW module separately

- Allows the identification of “troubled” modules through a sortable HTML table

- Gives instruction statistics and produces detailed graphs to make analysis easier

- It requires 21 identical cmsRun’s (no multiple sets of events are used → more accurate results), but it can be parallelized so (using 7 cores): time = ~3 runs

- Code outside modules is not monitored (framework)
<table>
<thead>
<tr>
<th>Total Cycles</th>
<th>Cycles Stalled</th>
<th>% of Cycles Stalled</th>
<th>CPI Ratio</th>
<th>Improvement Margin</th>
<th>iFactor</th>
<th>L2 Miss Impact</th>
<th>% of Total Stalls</th>
<th>L2 Hit Impact</th>
<th>% of Total Stalls</th>
<th>L1 DTLB Miss Impact</th>
<th>% of Total Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>205757078</td>
<td>73002049</td>
<td>35.5%</td>
<td>1.13</td>
<td>12.3562%</td>
<td>2.2116</td>
<td>860596</td>
<td>2.5%</td>
<td>13252348</td>
<td>38.1%</td>
<td>1088361</td>
<td>3.1%</td>
</tr>
<tr>
<td>107164496</td>
<td>41917192</td>
<td>39.1%</td>
<td>1.22</td>
<td>6.5664%</td>
<td>1.0426</td>
<td>725514</td>
<td>3.7%</td>
<td>7881293</td>
<td>39.9%</td>
<td>667089</td>
<td>3.4%</td>
</tr>
<tr>
<td>87458073</td>
<td>36272929</td>
<td>41.5%</td>
<td>1.20</td>
<td>5.3403%</td>
<td>0.8352</td>
<td>740136</td>
<td>3.9%</td>
<td>9606730</td>
<td>50.0%</td>
<td>1044767</td>
<td>5.4%</td>
</tr>
<tr>
<td>87257150</td>
<td>36845111</td>
<td>42.2%</td>
<td>1.22</td>
<td>5.3479%</td>
<td>0.8539</td>
<td>1074574</td>
<td>5.6%</td>
<td>9509844</td>
<td>49.2%</td>
<td>1085790</td>
<td>5.6%</td>
</tr>
<tr>
<td>69623622</td>
<td>27098126</td>
<td>38.9%</td>
<td>1.18</td>
<td>4.2375%</td>
<td>0.6077</td>
<td>938659</td>
<td>7.5%</td>
<td>4538021</td>
<td>36.3%</td>
<td>438571</td>
<td>3.5%</td>
</tr>
<tr>
<td>56071416</td>
<td>20759387</td>
<td>37.0%</td>
<td>1.15</td>
<td>3.3863%</td>
<td>0.4269</td>
<td>523417</td>
<td>5.2%</td>
<td>3706486</td>
<td>36.9%</td>
<td>321848</td>
<td>3.2%</td>
</tr>
<tr>
<td>53481805</td>
<td>15844810</td>
<td>29.6%</td>
<td>0.88</td>
<td>2.9479%</td>
<td>0.2833</td>
<td>284279</td>
<td>2.6%</td>
<td>4416432</td>
<td>41.0%</td>
<td>580631</td>
<td>5.4%</td>
</tr>
<tr>
<td>40712678</td>
<td>17601276</td>
<td>43.2%</td>
<td>1.25</td>
<td>2.5106%</td>
<td>0.3449</td>
<td>947122</td>
<td>9.7%</td>
<td>4447021</td>
<td>45.8%</td>
<td>560107</td>
<td>5.8%</td>
</tr>
<tr>
<td>40508212</td>
<td>16633763</td>
<td>41.1%</td>
<td>1.20</td>
<td>2.4735%</td>
<td>0.3225</td>
<td>533202</td>
<td>5.6%</td>
<td>4506761</td>
<td>47.6%</td>
<td>535283</td>
<td>5.7%</td>
</tr>
<tr>
<td>27315830</td>
<td>11298221</td>
<td>41.4%</td>
<td>1.25</td>
<td>1.6858%</td>
<td>0.2158</td>
<td>459206</td>
<td>8.4%</td>
<td>2069060</td>
<td>37.7%</td>
<td>194246</td>
<td>3.5%</td>
</tr>
<tr>
<td>25820246</td>
<td>11264500</td>
<td>43.6%</td>
<td>1.22</td>
<td>1.5825%</td>
<td>0.2049</td>
<td>1531928</td>
<td>24.9%</td>
<td>1948879</td>
<td>31.7%</td>
<td>209941</td>
<td>3.4%</td>
</tr>
<tr>
<td>25708037</td>
<td>10323668</td>
<td>40.2%</td>
<td>0.98</td>
<td>1.4793%</td>
<td>0.1753</td>
<td>2703393</td>
<td>44.7%</td>
<td>1318111</td>
<td>21.8%</td>
<td>244025</td>
<td>4.0%</td>
</tr>
<tr>
<td>22901445</td>
<td>10618268</td>
<td>46.4%</td>
<td>1.29</td>
<td>1.4250%</td>
<td>0.1941</td>
<td>1720125</td>
<td>30.4%</td>
<td>1714358</td>
<td>30.3%</td>
<td>185592</td>
<td>3.3%</td>
</tr>
</tbody>
</table>
BACKUP: Single Module Graphs

EcalRawToRecHitProducer_hltEcalRecHitAll - CYCLES: 25708037 - STALLED: 40.2% - CPI: 0.98

Stalls composition
- L2 miss impact: 44.172%
- L2 hit impact: 21.537%
- L1 dttb miss impact: 3.987%
- LCP stalls impact: 1.216%
- Store-fwd stalls impact: 29.088%

Store-forward stalls composition
- Loads blocked by unknown address store impact: 80.790%
- Loads overlapped with stores impact: 12.310%
- Loads spanning across cache line impact: 6.900%

Instruction type (ITLB miss rate = 0.01%)
- Loads: 25.103%
- Stores: 13.753%
- Branches: 16.462%
- SIMD Computational: 0.000%
- Other: 44.682%

Mispredicted branches = 0.01486
This value is the fraction of branches that were mispredicted.
BACKUP: Modular Symbol Level Analysis

- Uses the Perfmon2 interface (libpfm) directly and analyses each CMSSW module separately

- **Sampling periods** are specific to each event in order to have reasonable measurements

- The list of modules is a HTML table sortable by number of samples of `UNHALTED_CORE_CYCLES`

- For each module the complete set of usual events (Cycle Accounting Analysis & others) is sampled

- Results of each module are presented in separate HTML pages in tables sorted by decreasing sample count
## BACKUP: The List of Modules

### Sampling Analysis Results

<table>
<thead>
<tr>
<th>#SAMPLES (UNHALTED_CORE_CYCLES)</th>
<th>MODULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>118696</td>
<td>CkfTrackCandidateMaker_hltL1NonIsoEgammaRegionalCkfTrackCandidates</td>
</tr>
<tr>
<td>69756</td>
<td>ElectronSeedProducer_hltL1NonIsoLargeWindowElectronPixelSeeds</td>
</tr>
<tr>
<td>66556</td>
<td>ElectronSeedProducer_hltL1NonIsoStartUpElectronPixelSeeds</td>
</tr>
<tr>
<td>61888</td>
<td>CkfTrackCandidateMaker_hltL1IsoEgammaRegionalCkfTrackCandidates</td>
</tr>
<tr>
<td>53719</td>
<td>PixelTrackProducer_hltPixelTracksForMinBias</td>
</tr>
<tr>
<td>43667</td>
<td>CkfTrackCandidateMaker_hltBLifetimeRegionalCkfTrackCandidatesStartupU</td>
</tr>
<tr>
<td>33560</td>
<td>CkfTrackCandidateMaker_hltCkfL1NonIsoLargeWindowTrackCandidates</td>
</tr>
<tr>
<td>32355</td>
<td>ElectronSeedProducer_hltL1IsoLargeWindowElectronPixelSeeds</td>
</tr>
<tr>
<td>30706</td>
<td>ElectronSeedProducer_hltL1IsoStartUpElectronPixelSeeds</td>
</tr>
<tr>
<td>27055</td>
<td>EcalRawToRecHitProducer_hltEcalRecHitAll</td>
</tr>
<tr>
<td>20868</td>
<td>L2TauNarrowConeIsolationProducer_hltL2TauNarrowConeIsolationProducer</td>
</tr>
</tbody>
</table>
### BACKUP: Table Example of a Module

<table>
<thead>
<tr>
<th>Samples</th>
<th>Percentage</th>
<th>Symbol Name</th>
<th>Library Name</th>
<th>Complete Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>3427</td>
<td>8.291196%</td>
<td>__ieee754_atan2</td>
<td>libm-2.5.so</td>
<td>__ieee754_atan2</td>
</tr>
<tr>
<td>1281</td>
<td>3.099219%</td>
<td>_int_malloc</td>
<td>libc-2.5.so</td>
<td>_int_malloc</td>
</tr>
<tr>
<td>1202</td>
<td>2.908088%</td>
<td>tan</td>
<td>libm-2.5.so</td>
<td>tan</td>
</tr>
<tr>
<td>908</td>
<td>2.196792%</td>
<td>match</td>
<td>libRecoLocalTrackerSiStripRecHitConverter.so</td>
<td>SiStripRecHitMatcher::match(SiStripRecHit2D const*, __gnu_cxx::__normal_iterator&lt;SiStripRecHit2D const*, std::vector&lt;SiStripRecHit2D const*, std::allocator&lt;SiStripRecHit2D const*&gt; &gt; &gt;, __gnu_cxx::__normal_iterator&lt;SiStripRecHit2D const* const*, std::vector&lt;SiStripRecHit2D const*, std::allocator&lt;SiStripRecHit2D const*&gt; &gt; &gt;, boost::function&lt;void ()(SiStripMatchedRecHit2D const&amp;,&amp;, G4 GeomDet const*, Vector3DBase&lt;float, GlobalTag&gt; const)&gt; const</td>
</tr>
<tr>
<td>899</td>
<td>2.175018%</td>
<td>fesetenv</td>
<td>libm-2.5.so</td>
<td>fesetenv</td>
</tr>
<tr>
<td>849</td>
<td>2.054049%</td>
<td>computeFullJacobian</td>
<td>libTrackingToolsAnalyticalJacobians.so</td>
<td>AnalyticalCurvilinearJacobian::computeFullJacobian(GlobalTrajectoryParameters const&amp;, Point3DBase&lt;float, GlobalTag&gt; const&amp;, Vector3DBase&lt;float, GlobalTag&gt; const&amp;, Vector3DBase&lt;float, GlobalTag&gt; const&amp;, double const&amp;)</td>
</tr>
<tr>
<td>568</td>
<td>1.374205%</td>
<td>JacobianCurvilinearToLocal</td>
<td>libTrackingToolsAnalyticalJacobians.so</td>
<td>JacobianCurvilinearToLocal::JacobiansCurvilinearToLocal(Surface const&amp;, LocalTrajectoryParameters const&amp;, MagneticField const&amp;)</td>
</tr>
<tr>
<td>566</td>
<td>1.369366%</td>
<td>localMomentum</td>
<td>libTrackingToolsTrajectoryState.so</td>
<td>BasicSingleTrajectoryState::localMomentum() const</td>
</tr>
<tr>
<td>566</td>
<td>1.369366%</td>
<td>__ieee754_exp</td>
<td>libm-2.5.so</td>
<td>__ieee754_exp</td>
</tr>
</tbody>
</table>
BACKUP: Structure and libraries

Start

Analysis Configuration

Performance Data Taking

Program Run

Performance Data Output

Performance Data Analysis

Browsable HTML results

End

zlib

libpng

libSDL

libSDL_ttf

Version with graphs
The Sampling Process

**For each algorithm...**

- **Start**
  - Sampler initialization
  - Start
  - Sampling starts
  - Algorithm executes its code
  - Buffer full?
    - Yes: Algorithm stops executing its code
    - No: Algorithm finished?
      - Yes: Samples dumped to output files
      - No: Sampler termination
  - Buffer flushes content to memory
  - Sampling stops
  - End

**End**
## BACKUP: Module Level Analysis Results Snapshot

### RESULTS:

Click for graphs...

<table>
<thead>
<tr>
<th>MODULE NAME</th>
<th>Total Cycles</th>
<th>Cycles Stalled</th>
<th>% of Cycles Stalled</th>
<th>CPI Ratio</th>
<th>Improvement Margin</th>
<th>iFactor</th>
<th>L2 Miss Impact</th>
<th>% of Total Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateOfflinePhotons</td>
<td>66099336</td>
<td>25827555</td>
<td>39.1%</td>
<td>1.27</td>
<td>7.9836%</td>
<td>1.5116</td>
<td>2786991</td>
<td>14.8%</td>
</tr>
<tr>
<td>FitForward</td>
<td>52049150</td>
<td>15468001</td>
<td>29.7%</td>
<td>0.99</td>
<td>5.8517%</td>
<td>0.7391</td>
<td>5391694</td>
<td>28.5%</td>
</tr>
<tr>
<td>FitMatch</td>
<td>48407093</td>
<td>14049725</td>
<td>29.0%</td>
<td>0.97</td>
<td>5.3973%</td>
<td>0.6471</td>
<td>4874571</td>
<td>27.4%</td>
</tr>
<tr>
<td>MuonIDAlg</td>
<td>44410064</td>
<td>15731374</td>
<td>35.4%</td>
<td>1.17</td>
<td>5.2587%</td>
<td>0.9655</td>
<td>4277709</td>
<td>20.0%</td>
</tr>
<tr>
<td>FitDownstream</td>
<td>40519279</td>
<td>12222142</td>
<td>30.2%</td>
<td>0.97</td>
<td>4.5191%</td>
<td>0.5449</td>
<td>3686451</td>
<td>25.1%</td>
</tr>
<tr>
<td>RichOfflineGPIDLLIt0</td>
<td>37708627</td>
<td>12480747</td>
<td>33.1%</td>
<td>1.06</td>
<td>4.3366%</td>
<td>0.4460</td>
<td>3594169</td>
<td>28.7%</td>
</tr>
<tr>
<td>FitSeedForMatch</td>
<td>36451853</td>
<td>10623706</td>
<td>29.1%</td>
<td>0.94</td>
<td>4.0237%</td>
<td>0.4471</td>
<td>3079995</td>
<td>24.0%</td>
</tr>
<tr>
<td>RefitSeed</td>
<td>35875307</td>
<td>10299942</td>
<td>28.7%</td>
<td>0.93</td>
<td>3.9507%</td>
<td>0.4273</td>
<td>2813026</td>
<td>22.4%</td>
</tr>
<tr>
<td>RichOfflineGPIDLLIt1</td>
<td>25363184</td>
<td>7987786</td>
<td>31.5%</td>
<td>1.12</td>
<td>2.9631%</td>
<td>0.2486</td>
<td>2810843</td>
<td>24.0%</td>
</tr>
<tr>
<td>SpdMon</td>
<td>21383173</td>
<td>5174171</td>
<td>24.2%</td>
<td>0.93</td>
<td>2.3552%</td>
<td>0.2267</td>
<td>911354</td>
<td>8.9%</td>
</tr>
<tr>
<td>FitVelo</td>
<td>21243866</td>
<td>5834260</td>
<td>27.5%</td>
<td>0.97</td>
<td>2.3692%</td>
<td>0.2037</td>
<td>933870</td>
<td>12.9%</td>
</tr>
</tbody>
</table>
**BACKUP: Improvement Margin and iFactor**

### iMargin (CPI reduction effects)

Data: `cur_CPI, exp_CPI, local_cyc_bef, glob_cyc_before`.

- `loc_imp_ratio = cur_CPI/exp_CPI`
- `loc_cyc_after = local_cyc_bef/loc_imp_ratio`
- `glob_cyc_after = glob_cyc_before – loc_cyc_before + loc_cyc_after`
- `improvement_margin = 100 – (glob_cyc_after/glob_cyc_before) * 100`

### iFactor (Improvability Factor)

Data: `simd_perc, missp_ratio, stalled_cycles`.

- `simd_factor = 1 – normalized(simd_perc)`
- `missp_factor = normalized(missp_ratio)`
- `stall_factor = normalized(stalled_cycles)`
- `iFactor = stall_factor * (simd_factor + missp_factor + stall_factor)`
BACKUP: “Vertical” vs. “Horizontal” cut

- For Gaudi and CMSSW we used a “horizontal” cut
- Geant4 doesn’t provide hooks for any horizontal cut
- Modular analysis through User Actions
- “Time division” instead of “Code division”
- Useful or not? maybe... taking particles, energies and volumes into consideration
- Moreover modular symbol analysis still provides “Code division”
BACKUP: Choice of granularity

- Different levels of granularity were considered (run, event, track and step) as each offered User Actions

- Step-level granularity was the final winner

- At each step the particle, its energy (at the beginning) and the physical volume that it is running through are used

- Interesting volumes (at any level in the geometry tree) are given through an input file and used in the results view

- Other volumes are labeled as “OTHER”

- Results are browsable by any of the above variables
BACKUP: “Total” vs. “Average” count

- **CMSSW** and **Gaudi** used **average counts** of performance events

- All **modules** were “used” the same number of times during a single execution

- No longer true in Geant4 steps since “modules” here are a combination of physics variables

- Therefore we chose to display **total counts** of all performance counters

- *Exception:* for the number of **UNHALTED_CORE_CYCLES** we provide both average and total counts
BACKUP: (1/3) How to use it?

• Unpack the following archive in your application directory:
  http://dkruse.web.cern.ch/dkruse/G4_pfm.tar.gz

• Add the following lines to your GNUmakefile (to link *libpfm*_):
  ```
  CPPFLAGS += -I/usr/include/perfmon
  EXTRALIBS += -lpfm -ldl -
   L/afs/cern.ch/sw/lcg/external/libunwind/0.99/x86_64-slc5-gcc43-opt/lib -lunwind
  EXTRALIBSSOURCEDIRS +=
   /afs/cern.ch/sw/lcg/external/libunwind/0.99/x86_64-slc5-gcc43-opt
  ```

• Edit the “RUN_CONFIG” attribute in the *pfm_config_arch.xml* file inserting the normal run command. Example:
  ```
  RUN_CONFIG="~/geant4/bin/Linux-g++/full_cms bench10.g4"
  ```

• Edit the *pvs.txt* file inserting the interesting physical volumes:
  ```
  CALO   MUON
  VCAL   BEAM
  TRAK
  ```
BACKUP: (2/3) How to use it?

• Add the following lines to your `main()` before `runManager->Initialize()`:

```cpp
#include "PfmSteppingAction.hh"
...
int base_arg_no = 2; //number of standard arguments including executable name
runManager->SetUserAction(new PfmSteppingAction(argv[base_arg_no],
    argv[base_arg_no+1], atoi(argv[base_arg_no+2]),
    (unsigned int)atoi(argv[base_arg_no+3]), (unsigned int)atoi(argv[base_arg_no+4]),
    argv[base_arg_no+5], atoi(argv[base_arg_no+6]),
    (unsigned int)atoi(argv[base_arg_no+7]), (unsigned int)atoi(argv[base_arg_no+8]),
    argv[base_arg_no+9], atoi(argv[base_arg_no+10]),
    (unsigned int)atoi(argv[base_arg_no+11]), (unsigned int)atoi(argv[base_arg_no+12]),
    argv[base_arg_no+13], atoi(argv[base_arg_no+14]),
    (unsigned int)atoi(argv[base_arg_no+15]), (unsigned int)atoi(argv[base_arg_no+16]),
    (unsigned int)atoi(argv[base_arg_no+17]), argv[base_arg_no+18],
    (bool)atoi(argv[base_arg_no+19])));
```

• Compile and link:

```bash
gmake
g++ -Wall -o create create_config_files_from_xml.cpp -lxerces-c

g++ -Wall -lz -o analyse pfm_gen_analysis.cpp
```
BACKUP: (3/3) How to use it?

• Create results directory:
  mkdir results

• Create python run script “G4perfmon_runs.py”:
  ./create pfm_config_nehalem.xml

• Run the application with the perfmon monitor:
  python G4perfmon_runs.py &

• Analyse the results (optionally generating csv file):
  ./analyse results/ --caa [--csv]

• Check your results using your favourite browser:
  firefox results/HTML/index.html
<?xml version="1.0" ?>

<PFM_CONFIG>
  <PROPERTIES NAME="GeneralAnalysis" RUN_CONFIG="hlt_HLT.py" OUTPUT_DIR="results/" />
  <CONFIG START_AT_EVENT="4" PARALLEL="0" />
  <EVENTS>
    <EVENT NAME="BR_INST_RETIRED:ALL_BRANCHES" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
    <EVENT NAME="ILD_STALL:ANY" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
    <EVENT NAME="MEM_INST_RETIRED:LOADS" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
    <EVENT NAME="MEM_INST_RETIRED:STORES" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
  </EVENT_SET>
  <EVENT_SET>
    <EVENT NAME="INST_RETIRED:ANY_P" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
    <EVENT NAME="ITLB_MISS_RETIRED" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
    <EVENT NAME="MEM_LOAD_RETIRED:DTLB_MISS" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
    <EVENT NAME="MEM_LOAD_RETIRED:L2_HIT" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
  </EVENT_SET>
  <EVENT_SET>
    <EVENT NAME="ARITH:CYCLES_DIV_BUSY" CMASK="0" INVMASK="0" SMPL_PERIOD="1000" />
  </EVENT_SET>
</EVENTS>
</PFM_CONFIG>