CMSSW Performance Monitoring on processors based on the Intel Core and Nehalem Microarchitectures

Daniele Francesco Kruse

March 24, 2011

Contents

Abstract

CPU clock frequency is not likely to be increased significantly in the coming years, and data analysis speed can be improved by using more processors or buying new machines, only if one is willing to change the programming paradigm to a parallel one. Therefore, performance monitoring procedures and tools are needed to help programmers to optimize existing software running on current and future hardware, without having to redesign it completely. Low level information from hardware performance counters is vital to spot specific performance problems slowing program execution. HEP software is often huge and complex, and existing tools are unable to give results with the required granularity. I will report on the approach I have chosen to solve this problem on CMSSW. It involves decomposing the application into parts and monitoring each one of them separately. Both counting and sampling methods are used to allow an analysis with the required custom granularity: from global level, up to the function level. A set of tools (based on perfmon2 - a software interface to hardware counters) has been developed and deployed.

This document is a report concerning the work done at CERN on performance monitoring of CMSSW, using a 4-way analysis based on hardware performance counters. I will present the Intel Core and Nehalem microarchitectures and the methodology used to analyse the software, the Cycle Accounting Analysis. Next I will describe the 4 analysis approaches used (Overall, Symbol, Module Level and Modular Symbol Level analysis) and their complementary usefulness in performance evaluation. A few paragraphs are dedicated to a basic explanation of the inner workings of the tools implemented.

Chapter 1

Introduction

The good old days, when Moore's Law guaranteed a stable and transparent computing performance gain each and every year, are over. Processor clock speed cannot be increased anymore and, even if it could, it would not help since memory is still far behind in terms of speed, it will not catch up with the processor in the near future and it would be the bottleneck. Programs are performance-greedy and, as they get larger and more complex, they require improved and faster hardware to run properly. The hardware improvements that are available today include: multiple processors, multiple cores and NUMA architectures. Although all these are very promising, they are definitely non-transparent for programmers, for at least two reasons: firstly programmers need to write multi-threaded code, secondly, as the limited hardware resources (caches, bus, main memory) are shared among cores and processors, programmers need to constantly monitor how their programs use these resources in order to avoid bottlenecks and to speed up performance. Writing efficient and correct multi-threaded code is non-trivial and will not be covered in this article. Instead we will focus on monitoring the performance of single-threaded programs, to find problems and inefficiencies in the code, to optimize it and to get the most out of today's hardware. This article and all the research done, concerns only the most recent Intel processor families *Core* and *Nehalem* [1, 2].

1.1 Motivation and Goal

CMS software is huge and complex and it is developed by hundreds of physicists and programmers often unaware of performance issues. As pointed out in the introduction, the reason comes from the fact that, for many years, expertise in this field was not required, because the ever faster hardware would

compensate for it. The software produced is therefore suboptimal in terms of efficiency and speed. Moreover, given the heterogeneous group of developers of HEP software and its large and complex structure (hundreds of libraries and thousands of classes), the job of the performance optimization teams is more difficult than ever. Our goal was then to develop tools and procedures, to help spotting the problems and finding the parts of code responsible for them, so that they could be solved eventually.

Chapter 2

The Microarchitectures

2.1 Intel Core Microarchitecture

This section will give the reader a very brief introduction to the Intel Core Microarchitecture in a way which is relevant to software performance monitoring and optimization.

2.1.1 Pipeline

Intel CORE Microarchitecture pipeline consists of:

In-order front end : fetches instruction streams from memory, with four instruction decoders to supply decoded instruction (μops) to the out-of-order execution core.

Out-of-order superscalar execution core : can issue up to six μops per cycle and reorder μops to execute as soon as sources are ready and execution resources are available.

In-order retirement unit : ensures the results of execution of μops are processed and architectural states are updated according to the original program order.

In the following we will present the main features of the front-end and the execution core.

2.1.2 In-order front end

The front ends supplies a stream of decoded instructions (i.e. μops) to a six-issue wide out-of-order engine. It is made of three components: the Branch

Prediction Unit (BPU), the Instruction Fetch Unit and the Instruction Queue and Decode Unit.

Branch Prediction Unit

The Branch Prediction Unit helps the instruction to fetch unit fetch the most likely instruction to be executed by predicting the various branch types: conditional, indirect, direct, call, and return. It uses dedicated hardware for each type. It enables speculative execution, and it improves its efficiency by reducing the amount of code in the "non-architected path" (code paths that the processor thought it should execute but then found out it should go in another path) to be fetched into the pipeline.

Instruction Fetch Unit

The Instruction Fetch Unit prefetches instructions that are likely to be executed, caches frequently-used instructions, and predecodes and buffers instructions, maintaining a constant bandwidth despite irregularities in the instruction stream.

Instruction Queue and Decode Unit

The Instruction Queue and Decode Unit decodes up to four instructions, or up to five with macro-fusion. The instruction queue is 18 instructions deep. It sits between the instruction predecode unit and the instruction decoders. It sends up to five instructions per cycle, and supports one macro-fusion per cycle. It also serves as a loop cache for loops smaller than 18 instructions, enabling some loops to be executed with both higher bandwidth and lower power.

2.1.3 Out-of-order superscalar execution core

The execution core of the Intel Core microarchitecture is superscalar and can process instructions out of order. When a dependency chain causes the machine to wait for a resource (such as a second-level data cache line), the execution core executes other instructions. This increases the overall rate of instructions executed per cycle (IPC). The execution core contains the following three major components: a Reservation Station, a Reorder Buffer and a Renamer.

Reservation station (RS)

Queues μops until all source operands are ready, schedules and dispatches ready μops to the available execution units. The RS has 32 entries. The initial stages of the out of order core move the μops from the front end to the ROB and RS. In this process, the out of order core carries out the following steps:

1. Allocates resources to μops.
2. Binds the μop to an appropriate issue port.
3. Renames sources and destinations of μops, enabling out of order execution.
4. Provides data to the μop when the data is either an immediate value or a register value that has already been calculated.

Renamer

Moves μops from the front end to the execution core. Architectural registers are renamed to a larger set of microarchitectural registers. Renaming eliminates false dependencies known as read-after-read and write-after-read hazards.

Reorder buffer (ROB)

Holds μops in various stages of completion, buffers completed μops, updates the architectural state in order, and manages ordering of exceptions. The ROB has 96 entries to handle instructions in flight.

2.2 Intel Nehalem Microarchitecture

This section will deal with the differences and improvements of the Intel Nehalem Microarchitecture with respect to the Intel Core Microarchitecture.

2.2.1 Front End

The instruction fetch unit can fetch up to 16 bytes of instructions each cycle from the instruction cache to the instruction length decoder (ILD). The instruction queue then buffers the instructions processed by the ILD and delivers up to four instructions in one cycle to the instruction decoder (ID). The

ID has three decoder units that can decode one simple instruction per cycle per unit. The other decoder unit can decode one instruction every cycle, either simple instruction or complex instruction made up of several μops.

In previous generations of Intel Core microarchitecture macro-fusion support was limited to very few instruction types and to 32-bit mode, in Nehalem instead, macro-fusion is supported in 64-bit mode, and many more instruction sequences are supported and converted into a single μop.

The hardware improves branch handling in several ways. Branch target buffer has been increased to increase the accuracy of branch predictions. Renaming is supported with return stack buffer to reduce mispredictions of return instructions in the code. Furthermore, in the Core microarchitecture the front end would be waiting to decode the proper instructions until resources were allocated to executing mispredicted code path. In the Nehalem microarchitecture new μops stream can start executing immediately as soon as the misprediction is detected.

2.2.2 Execution Engine

The out-of-order engine supports up to 128 micro-ops in flight. Each micro-ops must be allocated with the following resources: an entry in the re-order buffer (ROB), an entry in the reservation station (RS), and a load/store buffer if a memory access is required.

The RS is expanded to 36 entry deep (compared to 32 entries in the previous generation). Like in the Core microarchitecture, it can dispatch up to six micro-ops in one cycle if the micro-ops are ready to execute.

2.2.3 Cache and Memory Subsystem

Nehalem microarchitecture contains an instruction cache, a first-level data cache and a second-level unified cache in each core. Each physical processor contains several processor cores and a shared collection of subsystems that are referred to as "uncore", including: a unified third-level cache shared by all cores in the physical processor and the Intel QuickPath Interconnect links and associated logic.

The L1 and L2 caches are writeback and non-inclusive. The shared L3 cache is writeback and inclusive, such that a cache line that exists in either L1 data cache, L1 instruction cache, unified L2 cache also exists in L3. This is to minimize snoop traffic between processor cores.

Nehalem microarchitecture implements two levels of translation lookaside buffer (TLB). The first level consists of separate TLBs for data and code. DTLB0 handles address translation for data accesses, it provides 64 entries

to support 4KB pages and 32 entries for large pages. The ITLB provides 64 entries (per thread) for 4KB pages and 7 entries (per thread) for large pages. The second level TLB (STLB) handles both code and data accesses for 4KB pages. It supports 4KB page translation operation that missed DTLB0 or ITLB. All entries are 4-way associative.

2.2.4 Load and Store operations

Nehalem microarchitecture provides a peak issue rate of one 128-bit load and one 128-bit store operation per cycle. It has also enlarged the buffers for load and store operations: 48 load buffers, 32 store buffers. It has improved the speed when dealing with unaligned memory access and store-forwarding for most address alignments.

2.2.5 System Software

Synchronization primitives using the Lock prefix execute with reduced latency than in previous microarchitectures. Transitions between a Virtual Machine (VM) and its supervisor (the VMM) have also been reduced in processors based on Nehalem.

2.2.6 Power Consumption

Nehalem is designed for low power consumption while the system idles, through the use of "C-states", i.e. different levels of idle states.

2.2.7 Hyper-Threading Technology

Hyper-Threading Technology (HT) provides two logical processors sharing most execution/cache resources in each core. The HT implementation in Nehalem is much better compared to previous generations of HT implementations because Nehalem has a wider execution engine, more functional execution units, it supports higher peak memory bandwidth, it has larger instruction buffers and replicates (or partitions) almost all the resources needed by the instructions of each hardware thread (replicated: register state, renamed return stack buffer, and large-page ITLB - partitioned: load buffers, store buffers, re-order buffers, and small-page ITLB) with the only exception of the execution units.

Chapter 3

The PMU and Perfmon

In what follows, we will understand what hardware-based performance counters are in detail, and we will show how we use them to extract important information on how the monitored application performs.

Performance monitoring can be defined as *the action of collecting information related to how an application or system performs* [7]. The aim of performance monitoring is to identify bottlenecks and remove them to improve software performance.

Hardware performance counters are the resource we use for application performance monitoring. But what are performance counters? All new micro-architectures include a special hardware unit called PMU or Performance Monitoring Unit that contains a set of registers called performance counters (2 for Core and 4 for Nehalem). Access to the PMU requires kernel support to read and write its privileged registers. These counters are able to detect and count certain micro-architectural events from several hardware sources, like the pipeline, system bus or cpu caches. These include events such as CPU cycles, cache references, misses, instruction TLB (Translation Lookaside Buffer) misses, system bus utilization, and other kinds of possible events and states of the processor. Such events provide facilities to characterize the interaction between programmed sequences of instructions and microarchitectural subsystems. In contrast to machine simulations, they are available on most of today's hardware and they do not require software to be modified or recompiled. When properly used, they incur in a very low overhead. There is an enormous amount of event types that can be counted, so a big effort has to be put on deciding which events we should count and how we should use them to understand the performance of an application.

In order to tell the PMU which events we want to monitor and when we want to start or stop monitoring, we use an API called *Perfmon2* [6]. This interface was developed and it is currently maintained by Stephane Eranian of

Google Corporation (though he did most of *Perfmon2* related work when he was working at HP). It is portable across many recent micro-architectures, it supports system-wide and per-thread monitoring and, besides counting events, it also supports sampling.

Figure 3.1 shows the layering of perfmon components. At the bottom we see the CPU Hardware that contains the PMU. *Perfmon2* interacts with the PMU using a patched Linux kernel. In fact the vanilla kernel does not include support for perfmon. The *Perfmon2* library (libpfm) is divided in two parts: architectural and generic. The architectural part is specific for the microarchitecture used in the machine (in our case Intel Core Microarchitecture), while the generic part provides a common interface to the user of the library. On top of *Perfmon2* library there are the user space applications that make use of the library. As said before, `pfmon` is such an application, but also the performance analysis tool that we developed for CMSSW is an other example.

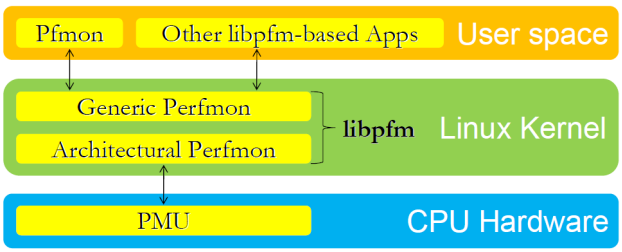


Figure 3.1: *Layering of perfmon components.*

Chapter 4

Cycle Accounting Analysis

The Cycle Accounting Analysis is a methodology to analyse the performance of an application and to find its weak points. It is specific for the Intel Core Microarchitecture and was developed by David Levinthal of Intel Corporation. It is also the recommended methodology of the Intel Architectures Optimization Reference Manual. Figure 4.1 illustrates the cycle decomposition schema of an application execution. According to it, the total execution time, i.e. the total execution cycles of an application, can be divided into cycles in which the front-end is issuing μops and cycles in which it is not. The cycles in which the front-end is issuing μops can be further divided into cycles which are retiring μops and cycles which are not. One example of μops issued but not retired is when branch misprediction occurs, μops which were issued and executed do not get retired since they belong to a speculative execution which did not prove correct eventually. So basically we have three possible types of cycles: cycles retiring μops, i.e. doing useful work, cycles issuing μops but not retiring them, i.e. doing useless work and cycles which are not issuing μops at all, i.e. stalled cycles doing no work. In the Core microarchiteture the stalled cycles can be further decomposed in 5 major components to better understand who is responsible for the lost cycles: level-2 cache misses, level-1 cache misses (but with a level-2 hit), instruction and data translation look-aside buffers misses (inside the level-1 cache), store-forwarding related stalls and finally stalls related to the use of length changing prefix instructions which involve the use of the slow decoder. Instead for the Nehalem microarchitecture there are more stall-related events that account for for the lost cycles. Nevertheless these can all be binned into the following categories: data load related stalls, floating point exceptions, cycles stalled due to long-latency divisions and/or square root operations executing, instruction fetching related stalls, stalls due to jumps and branches [4].

The aim of software optimization is therefore:

1. To bring the stalled cycles close to 0% by improving code and data locality for example.

2. To do the same for cycles that are not retiring μops by minimizing branches or use more predictable branching.

3. To reduce the number of cycles which are retiring μops by using vector instructions where possible, and using faster and more efficient algorithms of course.

Doing so will result in fewer total cycles and therefore a faster application.

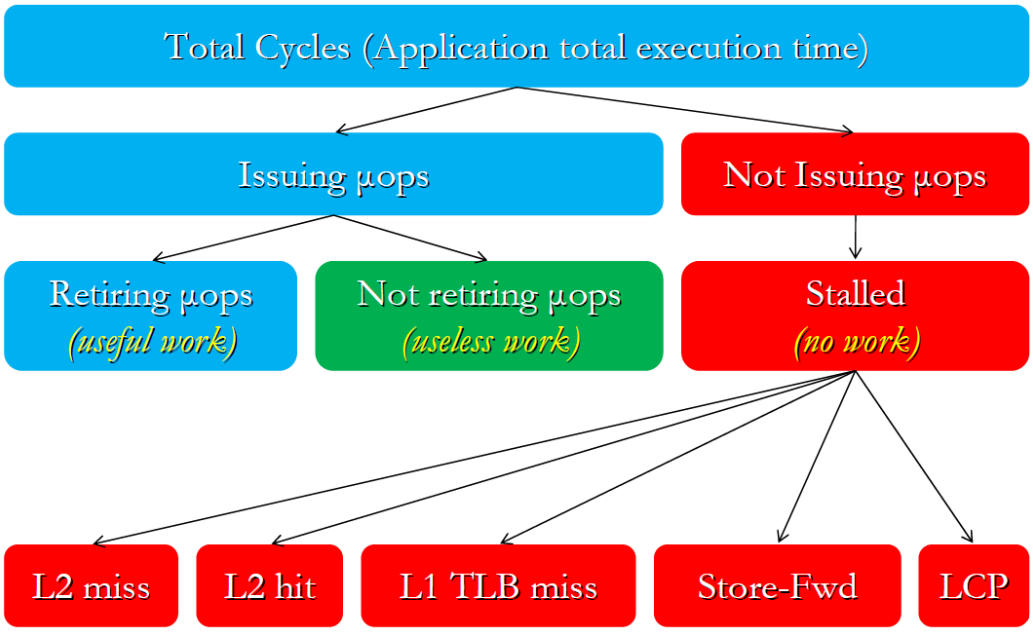


Figure 4.1: *Cycle Accounting Analysis - Cycle Decomposition.*

To give a more detailed overview of Cycle Accounting Analysis, let us now see how it works in detail for Intel Core microarchitecture, since it is slightly easier than in Nehalem. The first and most important event count for performance evaluation is the number of total clock cycles needed by an application to terminate successfully its execution. This metric can be measured by the event CPU_CLK_UNHALTED.CORE (aka UNHALTED_CORE_CYCLES). It is the most important metric because it is the only one that has to be considered at the end of any optimization process to see if we did a good job or not.

All other events must be taken into account with the only aim to eventually reduce the UNHALTED_CORE_CYCLES event count.

As all the cycles used by an application can be (roughly) divided into cycles not issuing μops and cycles issuing μops, we need a way to calculate them using performance counters. It turns out that the only event we need to monitor in this case is called RS_UOPS_DISPATCHED, that is the number of μops dispatched by the Reservation Station (RS) into the various execution ports. One useful feature of Intel Core Microarchitecture's PMU is the counter mask (aka CMASK): when it is set to something larger than zero, say n, it tells the counters to count the number of cycles (and NOT the number of events) during which the event monitored has occurred at least n times. Therefore if we want to know, for instance, how many cycles the Reservation Station dispatched at least 2 μops (in one single cycle) we would monitor the RS_UOPS_DISPATCHED setting the CMASK to 2. In our case we just need to know how many cycles the Reservation Station dispatched any number (bigger than 0) of μops so we will set our CMASK to 1. So we have that:

$$Cycles\_issuing\_\mu ops = \text{RS\_UOPS\_DISPATCHED } (\text{CMASK} = 1)$$

Another interesting feature of Intel's PMU is the INV bit, that defaults to 0 but can also be set to 1. When the INV bit is set to 1 (and the CMASK is set to some n bigger than 0), cycles are counted only when the event monitored occurs less than n times. So, in our case, since we need to know how many cycles the RS did NOT issue ANY μops, we also need to set the CMASK to 1 (because we are still counting cycles, not μops) and the INV to 1; therefore:

$$Cycles\_not\_issuing\_\mu ops = \text{RS\_UOPS\_DISPATCHED } (\text{CMASK} = 1 \;\&\&\; \text{INV} = 1)$$

So the total number of cycles can be expressed as:

$$Total\_cycles \cong Cycles\_issuing\_\mu ops + Cycles\_not\_issuing\_\mu ops$$

There's no equal sign there because there are few situations that are not properly considered in this analysis, such as whether the RS is full or empty, or transient situations of RS being empty but some in-flight μops are getting retired. Nevertheless the following equation should hold within a (small) error:

$$\textbf{UNHALTED\_CORE\_CYCLES} \cong \textbf{RS\_UOPS\_DISPATCHED}\ (\textbf{CMASK} = 1) + \textbf{RS\_UOPS\_DISPATCHED}\ (\textbf{CMASK} = 1\ \&\&\ \textbf{INV} = 1)$$

The μops that are issued for execution are not necessarily retired. This happens when the μops are part of a speculative execution that ends up being wrong: mispredicted branching is a good example. Those μops that do not reach retirement do not help forward progress of program execution. Therefore the number of *Cycles_issuing_μops* can be further decomposed into *Cycles_non_retiring_μops* and *Cycles_retiring_μops*. Unfortunately there's no event capable of measuring the number of *Cycles_non_retiring_μops*. We will derive this metric from available performance events, and several assumptions.

We define μops_rate as:

$$\mu ops\_rate = \frac{Dispatched\_\mu ops}{Cycles\_issuing\_\mu ops}$$

where the quantity *Dispatched_μops* can be measured with the event **RS_UOPS_DISPATCHED** (without **CMASK** and **INV**). Thus:

$$\mu ops\_rate = \frac{\textbf{RS\_UOPS\_DISPATCHED}}{\textbf{RS\_UOPS\_DISPATCHED}\ (\textbf{CMASK}{=}1)}$$

Next we define the total number of μops retired as:

$$Retired\_\mu ops = \textbf{UOPS\_RETIRED.ANY} + \textbf{UOPS\_RETIRED.FUSED}$$

Next we approximate the number of non-retiring μops by:

$$Non\_retired\_\mu ops = Dispatched\_\mu ops - Retired\_\mu ops$$

Thus finally,

$$Cycles\_non\_retiring\_\mu ops = \frac{Non\_retired\_\mu ops}{\mu ops\_rate}$$

The number of cycles retiring μops is easier and can be calculated as:

$$Cycles\_retiring\_\mu ops = \frac{Retired\_\mu ops}{\mu ops\_rate}$$

We also define the number of cycles stalled as:

$$Cycles\_stalled = Cycles\_not\_issuing\_\mu ops$$

Therefore:

$$Cycles\_stalled = \textsc{rs\_uops\_dispatched}\ (\textsc{cmask} = 1\ \&\&\ \textsc{inv} = 1)$$

This methodology does not take into account situations where retiring μops and non-retiring μops may be dispatched in the same cycle into the Out-Of-Order (OOO) engine. Nevertheless this scenario does not occur very often and the method used finds results that are a very good approximation of what happens in reality. So finally we have that the three calculated components should sum up to the total number of cycles, i.e:

$$Total\_cycles \cong Cycles\_non\_retiring\_\mu ops + Cycles\_retiring\_\mu ops + Cycles\_stalled$$

So, for optimization purposes we have to keep in mind that:

- If the contribution from *Cycles_non_retiring_μops* is high, focusing on code layout and reducing branch mispredictions will be important.
- If the contribution from *Cycles_stalled* is high, additional drill-down may be necessary to locate bottlenecks that lie deeper in the microarchitecture pipeline.
- If the contributions from *Cycles_non_retiring_μops* and *Cycles_stalled* are both insignificant, the focus of performance tuning should be directed to code vectorization or other techniques to improve retirement throughput of hot spots.

We should now understand what part of the architecture is stressed by our program's execution and is therefore causing the stalled cycles that we just calculated. One thing to note at this time is that events that cause stalls can be counted using the PMU, but the count obtained is not the number of cycles lost (stalled) caused by the event. I will therefore use the concept of *impact* when talking about number of cycles stalled due a particular kind of event. These are easily obtained by multiplying the cycle penalty of a

certain kind of event (number of cycles stalled caused by that event) by the number of events (of the same kind) counted. The following items discuss several common stress points of the microarchitecture:

Level-2 Cache Miss Impact The Intel Core Microarchitecture has a two-level caching system, meaning that a miss at the second level involves an access to system memory. The latency of system memory varies with different chipsets, but it is generally in the order of more than one hundred cycles. Server chipsets tend to exhibit longer latency than desktop chipsets. The number L2 cache miss references can be measured by **MEM_LOAD_RETIRED:L2_LINE_MISS**. An estimation of overall L2 miss impact calculated by multiplying system memory latency by the number of L2 misses is only an approximation because it ignores the OOO engine's ability to handle multiple outstanding load misses:

$$L2\_miss\_impact \cong \textbf{MEM\_LOAD\_RETIRED:L2\_LINE\_MISS} * system\_memory\_latency$$

Level-2 Cache Hit Impact When a Level-1 Cache Miss occurs, it does not necessarily mean that the processor will find the data on the second level cache. It may happen that the data required is missing from the second level cache as well. Therefore, the number of L2 hits can be measured by the difference between the number of Level-1 Data Cache Misses and Level-2 Cache misses, i.e:

$$Level\_2\_Cache\_Hits = \textbf{MEM\_LOAD\_RETIRED:L1D\_LINE\_MISS} - \textbf{MEM\_LOAD\_RETIRED:L2\_LINE\_MISS}$$

As in the previous case to obtain the impact we have to multiply this quantity by the Level-2 Cache access latency:

$$L2\_hit\_impact \cong Level\_2\_Cache\_Hits * Level\_2\_Cache\_latency$$

This formula, just like the one above does not take into account the OOO engine's ability to handle multiple outstanding load misses.

L1 DTLB Miss Impact Another cause of CPU stalls are Data Translation Look-aside Buffer (DTLB) Misses that occur in the Level-1 Cache. The number of misses is calculated using **MEM_LOAD_RETIRED:DTLB_MISS**. Therefore:

$$DTLB\_miss\_impact \cong \textbf{MEM\_LOAD\_RETIRED:DTLB\_MISS} * DTLB\_miss\_cycle\_penalty$$

LCP Impact LCP stands for Length-Changing Prefix. When instructions of this type are fetched they require the use of the slow instruction decoder. The event **ILD_STALL** measures the number of times the slow decoder was triggered, so:

$$LCP\_impact \cong \textbf{ILD\_STALL} * LCP\_cycle\_penalty$$

Store Forwarding Stall Impact When a store forwarding situation does not meet address or size requirements imposed by hardware, a stall occurs. The delay varies for different store forwarding stall situations. Consequently, there are several performance events that provide fine-grain specificity to detect different store-forwarding stall conditions. Three components will be analysed:

A load blocked by preceding store to unknown address can be measured by the event **LOAD_BLOCK:STA**. So:

$$Load\_block\_sta\_impact \cong \textbf{LOAD\_BLOCK:STA} * Load\_block\_sta\_cycle\_penalty$$

The event **LOAD_BLOCK:OVERLAP_STORE** counts the number of load operations blocked because of an actual data overlap with a preceding store, or because of an ambiguous overlap from page aliasing in which the load and a preceding store have the same offset but into different pages. We have that:

$$Load\_block\_overlap\_store\_impact \cong \textbf{LOAD\_BLOCK:OVERLAP\_STORE} * Load\_block\_overlap\_store\_cycle\_penalty$$

A load spanning across cache line boundary can be measured by the event **LOAD_BLOCK:UNTIL_RETIRE**. So:

$Load\_block\_until\_retire\_impact \cong$ **LOAD_BLOCK:UNTIL_RETIRE** $*$ $Load\_block\_until\_retire\_cycle\_penalty$

So we have that these three contributions sum up to the total numbers of cycles lost due to Store Forwarding mechanism problems:

$Store\_forwarding\_stall\_impact \cong Load\_block\_sta\_impact +$
$Load\_block\_overlap\_store\_impact +$
$Load\_block\_until\_retire\_impact$

These constants were either directly specified in the Intel manuals or calculated by means of dedicated programs for the 8-core Intel(R) Xeon(R) CPU E5430 @ 2.66GHz.

In principle the sum of these five stalls contributions should give a result very close to the total number of stalled cycles calculated before:

$Cycles\_stalled \cong L2\_miss\_impact + L2\_hit\_impact +$
$DTLB\_miss\_impact + LCP\_impact + Store\_forwarding\_stall\_impact$

Anyway this approach has a few problems: first of all it implies a simplification since other kinds of stalls may occur besides the 5 categories we saw. Secondly the impact in terms of cycles lost for each stall event may have an error depending on the particular state the machine is at the moment in which the event occurs, for instance sometimes a L2 miss may cause a 160 cycle dalay other times a 250 cycle delay (we have used an average value of 201). Third, sometimes the sum of all impacts of different stalls exceeds the total number of cycles not issuing μops, meaning that their impact was overestimated or that some of them overlap. Some other times the sum is a little smaller than the total number of cycles not issuing μops, meaning that their impact was underestimated or that another kind of stall occurred and was not taken into account. Moreover there are several components which cannot be counted reliably on the Intel Core Microarchitecture. These fall into three main classes: stalls due to instruction starvation, stalls due to dependent chains of multi-cycle instructions (other than divide) and stalls

related to Front Side Bus saturation. Finally, almost all event counts are approximations of real events, although they are very good approximations since the error is typically below 3%. Nevertheless, although quantities may be over or underestimated they give a good insight as to which are the main problems to work on within a particular application.

Chapter 5

A 4-way Analysis

There are 4 approaches that have been used to analyse CMSSW. These approaches are depicted in Figure 5.1. The first one is the "Overall Analysis" that evaluated the performance of CMSSW as a whole, counting events regarding complete executions of `cmsRun`s. The second one is the "Symbol Level Analysis", it was also carried out using `pfmon` and analysed CMSSW as a whole, but this time I had to use the sampling capabilities of `pfmon` to identify the symbol names. The third one, called "Module Level Analysis", had to be implemented as a service of CMSSW, because we wanted to analyse how each module is performing and therefore we needed hooks to start counting at the beginning of modules and to stop counting when the modules finished executing. This third approach gave some interesting results but still we needed something that showed how single functions perform inside modules. Therefore a fourth approach was used and was implemented using a service as well. This approach, called "Modular Symbol Level Analysis", used the sampling features of the *Perfmon2* library and led to a deeper and more useful analysis of modules.

5.1 Overall Analysis

The Overall Analysis reports the performance of CMSSW as a whole, it uses `pfmon` as the monitoring tool and it is based on the Cycle Accounting Analysis described before. The Overall analysis is good for showing the overall performance of the software and check if changes did in fact improve it or not. It is also good to identify generic problems such as too many cycles stalled or too many branch mispredictions, but it is not good enough to locate inefficient parts nor to identify bad programming practices. In order to address these issues we need other kinds of analysis.

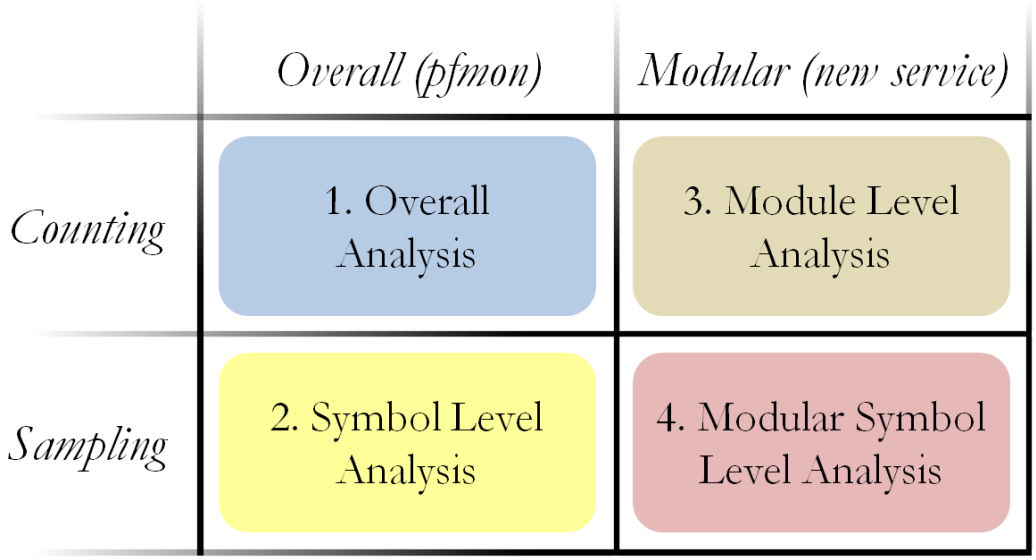


Figure 5.1: *The 4-way Analysis.*

The Overall Analysis is carried out using a bash script and an analysis program. The script is very simple and it is responsible for generating the raw performance data using `pfmon` as the performance monitoring tool. Inside this script `pfmon` counts the 13 events necessary to perform the Cycle Accounting Analysis and puts the results inside simple text files. One thing to note at this point is that not all event counts are stable: while some of them (like number of INSTRUCTIONS_RETIRED) are fairly stable across different runs, others (like number of UNHALTED_CORE_CYCLES) change quite a bit (around 3%) and may require multiple runs to get a reasonable average. That's why the UNHALTED_CORE_CYCLES event, is counted 5 times in 5 different runs while all the other events needed are counted just once. In a single `pfmon` run, up to 4 counters are used simultaneously: 2 fixed (UNHALTED_CORE_CYCLES and INSTRUCTIONS_RETIRED) and 2 programmable. Next, the output files of the script are passed to the analysis program. This program is responsible for extracting the data from the files, calculating the analysis metrics, and finally creating the user-friendly histograms that show the cycle composition at two stages (at issue port and at OOO engine) and the two-level stalled cycles decomposition: general stalls and store-forwarding stalls. In Figure 5.2 you have a self-descriptive sample output of the analysis program.

In order to create the graphs, I created a library called "dk_graph" that is based on the SDL (Simple DirectMedia Layer) library and on two of its

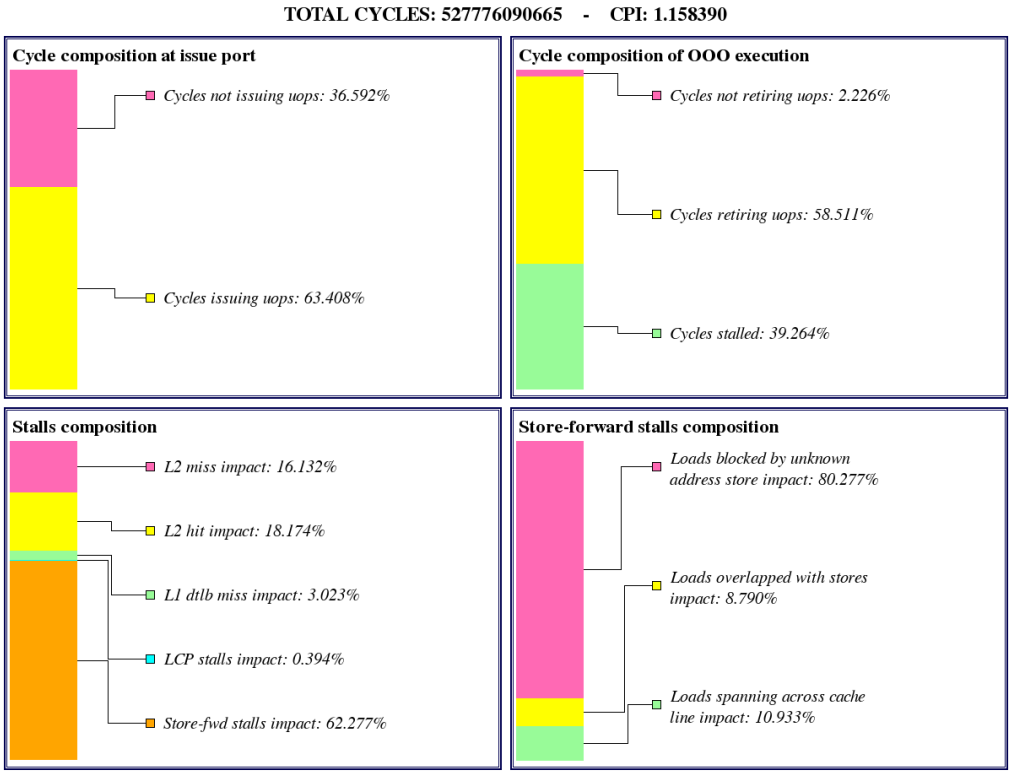


Figure 5.2: *Overall Analysis of CMSSW - Cycle Decomposition.*

plugins: the PNG (for the output format) and TTF (for the True-Type Fonts display) libraries. The "dk_graph" library is capable of creating two kinds of graphs:

BAR This kind is useful for viewing component percentages of a whole. The bar is divided into colored parts, each of them proportional to its contribution to the whole. Colours are selected automatically for a better view.

REF This kind is useful when you need to display a certain value with respect to a fixed low, medium and high reference values, allowing you to set explanations for those three possibilities. Colors are green for low values, yellow for medium values and red for high values.

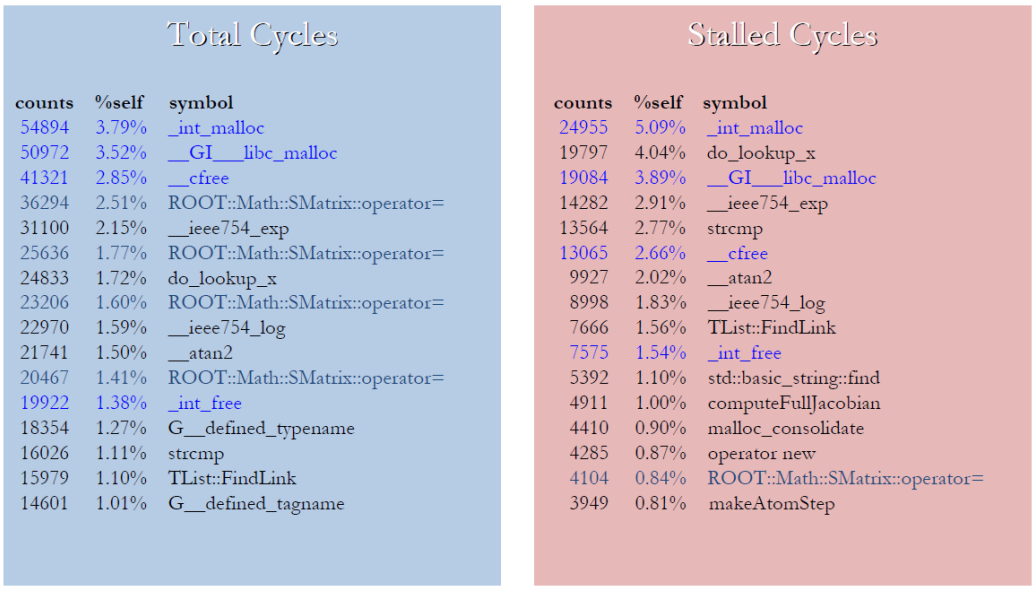
The library allows you to create PNG files out of your graphs or just display them of the screen. It can handle up to 12 graphs per view and each view is optimized for the number of graphs chosen (the smaller the better).

5.2 Symbol Analysis

Knowing that there is a high percentage of cycles stalled during a Level-2 Cache miss, is useful because it highlights the fact that there is a problem, unfortunately it doesn't say where the problem is and how to fix it. Therefore we need some other kind of analysis with a finer granularity. One of the features, already discussed, of `pfmon`, is sampling of events. Sampling of event counters works as follows: the event is counted normally, but when the count reaches a certain amount (the sampling period specified as command line argument to `pfmon`), the sampling buffer records the address of the instruction that was being executed in that moment; these addresses are then resolved into their containing functions (symbols), so that it is possible to find out how many times the sampling period ended exactly in this or that function. This is more or less a Montecarlo method to characterize how large a function is considering the specified event as the metric. So func_a may be larger than func_b considering UNHALTED_CORE_CYCLES, but it may be smaller considering INSTRUCTIONS_RETIRED, in this case this may suggest that func_b does its work in a more efficient way. But, as we will see, much more information can be extracted from sampling applications.

I called this kind of analysis "Symbol Analysis", as it allows the identification of symbols (or functions) in which certain events occur. It uses `pfmon` sampling capabilities and it's useful to identify bad programming practices and certain function behaviour (good and bad). As anticipated, in CMSSW's case (see Figure 5.3), this analysis shows something more than just the worst or best functions, or the functions with lowest cache misses and branch misprediction. It shows a bad programming practice, that is heavy usage of dynamic memory (de)allocation. Usually this practice leads to poor performance as calls to "malloc" and "free" primitives are expensive and should be used cautiously, although this is nothing new to the performance community, since dynamic memory usage has been shown to be a problem for CMSSW by many people with many different tools.

However the symbol analysis is not good for everything. It would be good enough for finding specific problems in the code if the application analysed were small. However CMSSW is huge and there are thousands of functions, so even if we now know that there are bad practices being used here and there, we now need to find where they are used. So, we really need to drill down inside the huge monster with finer granularity to find out how its building blocks behave as far as performance is concerned. To enter this next level, there's no help coming from `pfmon`, but a huge help is coming from its underlying library, *Perfmon2* and from CMSSW framework structure.



Total Cycles

counts	%self	symbol
54894	3.79%	_int_malloc
50972	3.52%	__GI___libc_malloc
41321	2.85%	__cfree
36294	2.51%	ROOT::Math::SMatrix::operator=
31100	2.15%	__ieee754_exp
25636	1.77%	ROOT::Math::SMatrix::operator=
24833	1.72%	do_lookup_x
23206	1.60%	ROOT::Math::SMatrix::operator=
22970	1.59%	__ieee754_log
21741	1.50%	__atan2
20467	1.41%	ROOT::Math::SMatrix::operator=
19922	1.38%	_int_free
18354	1.27%	G__defined_typename
16026	1.11%	strcmp
15979	1.10%	TList::FindLink
14601	1.01%	G__defined_tagname

Stalled Cycles

counts	%self	symbol
24955	5.09%	_int_malloc
19797	4.04%	do_lookup_x
19084	3.89%	__GI___libc_malloc
14282	2.91%	__ieee754_exp
13564	2.77%	strcmp
13065	2.66%	__cfree
9927	2.02%	__atan2
8998	1.83%	__ieee754_log
7666	1.56%	TList::FindLink
7575	1.54%	_int_free
5392	1.10%	std::basic_string::find
4911	1.00%	computeFullJacobian
4410	0.90%	malloc_consolidate
4285	0.87%	operator new
4104	0.84%	ROOT::Math::SMatrix::operator=
3949	0.81%	makeAtomStep

Figure 5.3: *Symbol Analysis of CMSSW - Total Cycles and Stalled Cycles.*

5.3 Module Level Analysis

CMSSW is made of a framework, an Event Data Model, and services needed by the simulation, calibration, alignment and reconstruction modules that process physical event data. The CMSSW physical event processing model consists of one executable, called `cmsRun`, and many plug-in modules which are managed by the framework. All the code needed in the physical event processing is contained within the modules, so that the bulk of total CMSSW execution time is spent during module code execution.

To monitor the performance of each module separately we need to find a way to raise a flag when the module starts executing and to pull it down when it terminates. Additionally a certain module may execute multiple times in one single run (with multiple physical events), that's why we need a mechanism to monitor it every time it executes during a `cmsRun`. One possible way (and, for some other piece of software, the only way) of doing this is to modify the source code of the modules to insert the code needed to start and stop monitoring their performance. Fortunately CMSSW offers a much more elegant and simple solution to this problem: services. These provide "hooks" that allow functions to be executed at a precise moment (a state transition of the application) during a normal `cmsRun`. There are hooks for almost any kind of state transition, but we only needed three:

postEventProcessing Called after all modules have processed a physical event

preModule Called before a module starts processing the current physical event

postModule Called after a module successfully processed the current physical event

So putting the performance monitoring code inside these hooks allowed me to generate counter results for each module separately. The corresponding CMSSW service created is called *PerfMonService*.

`pfmon` could not be used in this case because we didn't want to monitor one single function (which is possible for `pfmon` enabling its trigger functionality) nor one single module (which would have been also possible using `pfmon` in conjunction with services). We needed to analyse every module, separately, in one single run. As discussed before, `pfmon` relies on an API called *Perfmon2* to program the PMU counters. So, in our case, a direct access to *Perfmon2* was necessary. Using it involves setting up a few configuration structures that are used by the library to program the counters, and calling the appropriate functions to start and stop counting in `preModule` and `postModule` hooks, respectively. The other hook, namely `postEventProcessing`, is only used to count the number of physical events already executed, so that the start of the monitoring can be delayed after a certain amount of physical events have already been processed, thus avoiding initialization pollution. This number is of course configurable by the end-user.

Being a service, *PerfMonService*, can be enabled or disabled within the python configuration file normally used by `cmsRun` users to do their analysis. To make things easier, in order to use the service and provide the specific parameters to it, one only needs to use a dedicated script to create the various configuration files out of the original one, and then run another python script with no additional parameters. This script will be responsible to start all the `cmsRun`s needed. The end-user will not notice the difference between a normal run and a monitored run, since the overhead of the performance monitoring tool is really low (tipically below 1%).

The number of events monitored during this analysis is 17 for Core and 41 for Nehalem. On Nehalem it's possible to monitor up to 4 events during each run, while Core is limited to 2 events. Moreover, the various runs may be executed in parallel on a multicore machine, with only minor and negligible differences in counting results (well actually there are some visible effects on bus-related event counts, but they are not used in our analysis). The output

of these runs is a series of simple text files (one for each event) containing raw counter data for each module. These files are then passed to the post-processing analyser which reads the data and converts them to useful and human-readable performance figures. To do so, it creates a HTML page with a sortable table containing all the performance counts of events and of derived events and other useful performance information for each module, so that problematic and inefficient modules are easily spotted.

In Figure 5.4, one can see the partial view of the output of the analysis in tabular form. The table includes much more information than shown in the picture and it is sortable by any column. By clicking on any module of interest, one can access the Modular Symbol Level Analysis described in the next section.

MODULE NAME	Total Cycles	Cycles Stalled	% of Cycles Stalled	CPI Ratio	L2 Miss Impact	% of Total Stalls
CkfTrackCandidateMaker_newTrackCandidateMaker	1405883341	684506320	48.7%	1.25	22756000	3.7%
ConversionTrackCandidateProducer_conversionTrackCandidates	1216890912	673825912	55.4%	1.11	6577600	0.8%
GsfTrackProducer_pixelMatchGsfFit	1148110826	260087145	22.7%	0.91	778800	0.2%
GsfTrackProducer_gsfPFtracks	618961542	127697046	20.6%	0.86	435200	0.2%
CaloMuonProducer_calomuons	552939456	189712951	34.3%	1.15	7865000	6.0%
CkfTrackCandidateMaker_thTrackCandidates	510372704	220202685	43.1%	1.39	10047600	5.8%
TrackProducer_preFilterFirstStepTracks	424368027	172101497	40.6%	1.28	6569200	6.0%
MuonIdProducer_muons	423798580	140229777	33.1%	1.22	5562800	5.6%
PFRecoTauProducer_pfRecoTauProducerHighEfficiency	417091298	300934845	72.2%	2.80	244313400	77.8%
SoftElectronProducer_btagSoftElectrons	373227607	122767413	32.9%	1.15	4807600	5.1%
PoolOutputModule_RECO	367355743	71736042	19.5%	0.79	8666200	8.4%

Figure 5.4: *Module Level Analysis of CMSSW - Partial view of performance information data table.*

5.4 Modular Symbol Level Analysis

Since `pfmon`, for its sampling functionality, takes advantage of the underlying *Perfmon2* sampling capabilities, we were also able to use this API for our sampling needs inside our service. Nevertheless a few design issues had to be taken into account: first of all, unlike `pfmon`, we needed to monitor ourselves (since *PerfMonService* is part of CMSSW indeed) and not some other external process, second the sampling buffer is limited. Therefore we needed to install an appropriate signal handler to get the data from the buffer and flush it when the buffer becomes full. Another issue was the symbol (func-

tion) name resolution problem: as we will see in the next paragraph, symbol resolution has to be done using a two-step process, one online and another offline. This is due to the fact that the second step is expensive in terms of execution time and uses external programs that might pollute the performance data of the ongoing `cmsRun` execution.

Figure 5.5 shows in detail how the first step works. When the `cmsRun` starts its execution, the service gets initialized since its constructor is called. Then, just before the first module starts executing, the `preModule` hook function of *PerfMonService* is called. During this time all the data structures and parameters needed are prepared and passed down to the *Perfmon2* API in order to program the counters to monitor the appropriate events with the right flags (CMASK and INV), and the sampling process starts. While the module is running, sampling data (i.e. code addresses) are collected inside the sampling buffer. When the buffer is full an interrupt is generated by the PMU and the corresponding signal is caught by the service, the module stops executing its code and the control passes to the signal handler that is in charge of: saving the sampling data to memory, flushing the sampling buffer, resuming sampling process and giving the control back to the module that was running before the buffer overflow. This loop goes on until a module terminates its execution. At this time the sampling stops and the data collected by the sampling buffer is saved to memory. This whole process reapeats for each module and for each physical event processed. When all the modules finished processing all the physical events, the huge amount of raw data collected goes through a first pass of translation to symbols. This is carried out in the service destructor that uses `IgHookTrace`'s `symbol()` and `toSymbol()` functions to find out (and save to gzipped files) temporary symbol names, and most importantly library names and offsets of symbols in libraries. These two pieces of information alone are usually sufficient to find out the names of the real symbols, as we will see.

The second step takes place offline, i.e. while `cmsRun` is not running anymore. The various library offsets and library names, after all symbolic links are resolved, are translated into symbol names using the `symbolByOffset()` method of the `FileInfo` class of the `IgProf` library (developed by Lassi Tuura and Giulio Eulisse). However the symbol names returned by that method are not easily readable by end-users, and must be therefore demangled using a specific function: `__cxa_demangle()` of the `cxxabi` library. When the symbols are correctly resolved their sample count needs to be updated because multiple pairs of libraries and offsets may resolve in the same symbol, and their sample counts need to be summed up. Therefore an object is created for each module containing the array of correctly resolved symbol names and sample counts. Now we have everything: symbol names, libraries

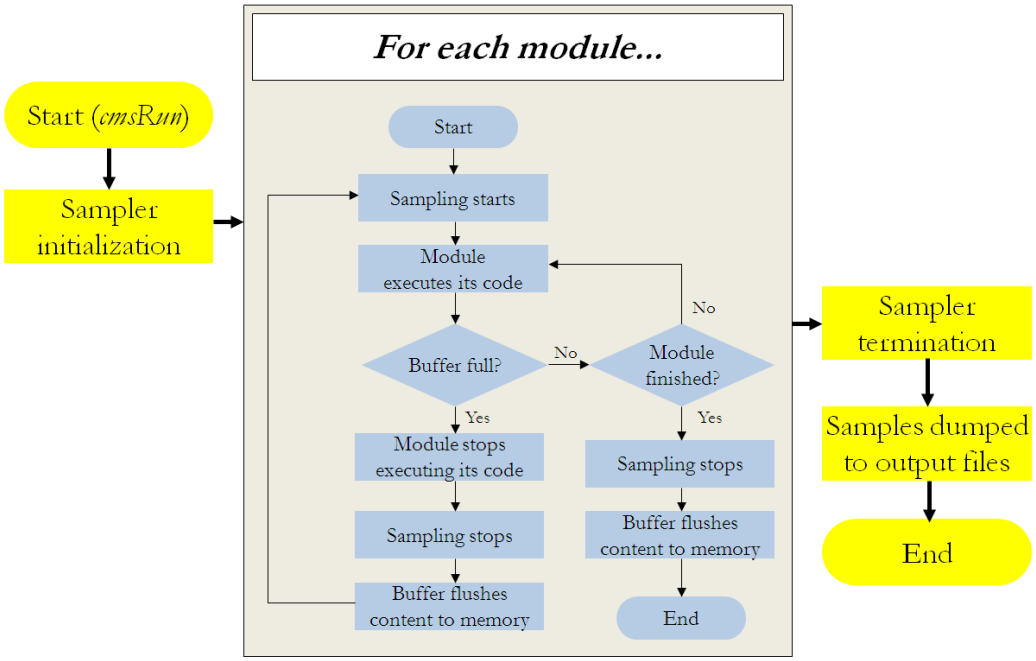


Figure 5.5: *Modular Symbol Level Analysis of CMSSW - Flow-graph of service event sampling algorithm.*

and sample counts. It is time to put this information in a useful and informative way for the end-user. To do so, a HTML page is created for each module and linked to the main table of the module level analysis. For each event there is a result table ordered by decreasing sample count (along with the percentage with respect to the total sample count of the module), featuring symbol name, symbol library and complete signature. Anyway not every symbol resolved is displayed, that would be an enormous amount of useless information. Only the symbols with the highest sample count get displayed (the current default is to display 20 symbols per microarchitectural event).

Figure 5.7 shows the details for the UNHALTED_CORE_CYCLES event, that is a table showing the functions responsible for the time spent during the execution of the module code.

Now the level of granularity has reached probably the limit. It is difficult to imagine how we could go further down, or if it is of any use at all. However the four kinds of analysis presented must not be considered mutually independent, or worse, an iterative refinement. Indeed modular symbol level analysis allows better and deeper code inspection when used in combination with module level analysis. This is due to the fact that first you need to find the troublesome and improvable modules using the module level analysis,

#SAMPLES (UNHALTED_CORE_CYCLES) ▲	MODULES
823141	CkfTrackCandidateMaker_newTrackCandidateMaker
436772	CkfTrackCandidateMaker_thTrackCandidates
402424	CkfTrackCandidateMaker_fourthTrackCandidates
352017	CkfTrackCandidateMaker_stepOneTrackCandidateMaker
351644	CkfTrackCandidateMaker_secTrackCandidates
237167	TrackProducer_preFilterZeroStepTracks
153092	TrackProducer_secWithMaterialTracks
136802	CkfTrackCandidateMaker_fifthTrackCandidates
85152	TrackProducer_thWithMaterialTracks
80976	SeedGeneratorFromRegionHitsEDProducer_newSeedFromPairs
79607	SeedGeneratorFromRegionHitsEDProducer_fourthPLSeeds
66401	TrackProducer_preFilterStepOneTracks

Figure 5.6: *Modular Symbol Level Analysis of CMSSW - Table listing modules by sample count.*

UNHALTED_CORE_CYCLES -- Total Samples: 823141 -- Sampling Period: 100000				
Samples	**Percentage**	**Symbol Name**	**Library Name**	**Complete Signature**
40542	4.92528%	__tan	libm-2.3.4.so	__tan
38761	4.70891%	match	libRecoLocalTrackerSiStripRecHitConverter.so	SiStripRecHitMatcher::match(SiStripRecHit2D const*, __gnu_cxx::__normal_iterator<SiStripRecHit2D const* const*, std::vector<SiStripRecHit2D const*, std::allocator<SiStripRecHit2D const*> > >, __gnu_cxx::__normal_iterator<SiStripRecHit2D const* const*, std::vector<SiStripRecHit2D const*, std::allocator<SiStripRecHit2D const*> > >, boost::function<void ()(SiStripMatchedRecHit2D const&)>&, GluedGeomDet const*, Vector3DBase<float, LocalTag>) const
33209	4.03442%	__atan2	libm-2.3.4.so	__atan2
30737	3.73411%	project	libRecoLocalTrackerSiStripRecHitConverter.so	SiStripRecHitMatcher::project(GeomDetUnit const*, GluedGeomDet const*, std::pair<Point3DBase<float, LocalTag>, Point3DBase<float, LocalTag> >, Vector3DBase<float, LocalTag>) const
29012	3.52455%	_int_malloc	libc-2.3.4.so	_int_malloc
16387	1.99079%	operator=	libTrackingToolsPatternTools.so	ROOT::Math::SMatrix<double, 5u, 5u, ROOT::Math::MatRepStd<double, 5u, 5u> >& ROOT::Math::SMatrix<double, 5u, 5u, ROOT::Math::MatRepStd<double, 5u, 5u> >::operator=<ROOT::Math::MatrixMulOp<ROOT::Math::SMatrix<double, 5u, 5u, ROOT::Math::MatRepStd<double, 5u, 5u> >, ROOT::Math::SMatrix<double, 5u, 5u, ROOT::Math::MatRepSym<double, 5u> >, double, 5u>, ROOT::Math::MatRepStd<double, 5u, 5u> > (ROOT::Math::Expr<ROOT::Math::MatrixMulOp<ROOT::Math::SMatrix<double, 5u, 5u, ROOT::Math::MatRepStd<double, 5u, 5u> >, ROOT::Math::SMatrix<double, 5u, 5u, ROOT::Math::MatRepSym<double, 5u> >, double, 5u>, double, 5u, 5u, ROOT::Math::MatRepStd<double, 5u, 5u> > const&)
14801	1.79811%	measurementPosition	libGeometryCommonTopologies.so	RadialStripTopology::measurementPosition(Point3DBase<float, LocalTag> const&) const

Figure 5.7: *Modular Symbol Level Analysis of CMSSW - View of symbols inside a module.*

then you need to find which functions are causing the problems using the modular symbol level analysis.

Chapter 6

Modular Analysis Models

This section deals with the details on how raw hardware counter data are transformed into derived information useful for performance evaluation, for Core and Nehalem microarchitectures in the module level analysis.

6.1 Core

Let us start with the definition of the following constants (specified in the Intel manuals):

- **CORE_L2_MISS_CYCLES** = 200
- **CORE_L2_HIT_CYCLES** = 14.5
- **CORE_L1_DTLB_MISS_CYCLES** = 10
- **CORE_LCP_STALL_CYCLES** = 6
- **CORE_UNKNOWN_ADDR_STORE_CYCLES** = 5
- **CORE_OVERLAPPING_CYCLES** = 6
- **CORE_SPAN_ACROSS_CACHE_LINE_CYCLES** = 20
- **EXPECTED_CPI** = 0.25

We monitor and we get the values of the following 17 microarchitectural events:

- **BRANCH_INSTRUCTIONS_RETIRED**
- **ILD_STALL**
- **INST_RETIRED:LOADS**
- **INST_RETIRED:OTHER**
- **INST_RETIRED:STORES**
- **INSTRUCTIONS_RETIRED**
- **LOAD_BLOCK:OVERLAP_STORE**

- **LOAD_BLOCK:STA**
- **LOAD_BLOCK:UNTIL_RETIRE**
- **MEM_LOAD_RETIRED:DTLB_MISS**
- **MEM_LOAD_RETIRED:L1D_LINE_MISS**
- **MEM_LOAD_RETIRED:L2_LINE_MISS**
- **MISPREDICTED_BRANCH_RETIRED**
- **RS_UOPS_DISPATCHED**
- **RS_UOPS_DISPATCHED CMASK=1 INV=1**
- **SIMD_COMP_INST_RETIRED:PACKED_SINGLE:PACKED_DOUBLE**
- **UNHALTED_CORE_CYCLES**

Using these two sets of raw data and the following formulas, we can derive important and useful performance information:

- $TotalCycles =$
  UNHALTED_CORE_CYCLES

- $StalledCycles =$
  RS_UOPS_DISPATCHED CMASK=1 INV=1

- $L2MissImpact =$
  MEM_LOAD_RETIRED:L2_LINE_MISS $*$ **CORE_L2_MISS_CYCLES**

- $L2HitImpact =$
  (**MEM_LOAD_RETIRED:L1D_LINE_MISS** $-$
  MEM_LOAD_RETIRED:L2_LINE_MISS) $*$ **CORE_L2_HIT_CYCLES**

- $L1DTLBMissImpact =$
  MEM_LOAD_RETIRED:DTLB_MISS $*$
  CORE_L1_DTLB_MISS_CYCLES

- $LCPStallsImpact =$
  ILD_STALL $*$ **CORE_LCP_STALL_CYCLES**

- $LoadsBlockedbyUnknownAddressStoreImpact =$
  LOAD_BLOCK:STA $*$
  CORE_UNKNOWN_ADDR_STORE_CYCLES

- $LoadsOverlappedwithStoresImpact =$
  LOAD_BLOCK:OVERLAP_STORE $*$
  CORE_OVERLAPPING_CYCLES

- $LoadsSpanningacrossCacheLinesImpact =$
  LOAD_BLOCK:UNTIL_RETIRE $*$
  CORE_SPAN_ACROSS_CACHE_LINE_CYCLES

- $StoreFwdStallsImpact =$
  $LoadsBlockedbyUnknownAddressStoreImpact +$
  $LoadsOverlappedwithStoresImpact +$
  $LoadsSpanningacrossCacheLinesImpact$

- $CountedStalledCycles =$
  $L2MissImpact +$
  $L2HitImpact +$
  $LCPStallsImpact +$
  $L1DTLBMissImpact +$
  $StoreFwdStallsImpact$

- $InstructionsRetired =$
  INSTRUCTIONS_RETIRED

- $ITLBMissRatein\% =$
  (**ITLB_MISS_RETIRED**/
  INSTRUCTIONS_RETIRED) $* 100$

- $BranchInstructions =$
  BRANCH_INSTRUCTIONS_RETIRED

- $LoadInstructions =$
  INST_RETIRED:LOADS

- $StoreInstructions =$
  INST_RETIRED:STORES

- $OtherInstructions =$
  INST_RETIRED:OTHER $-$
  SIMD_COMP_INST_RETIRED:PACKED_SINGLE:PACKED_DOUBLE $-$
  BRANCH_INSTRUCTIONS_RETIRED

- $\%ofMispredictedBranches =$
  (**MISPREDICTED_BRANCH_RETIRED**/
  BRANCH_INSTRUCTIONS_RETIRED) $* 100$

- $PackedSIMDComputationalInstructions =$
  SIMD_COMP_INST_RETIRED:PACKED_SINGLE:PACKED_DOUBLE

- $CountedInstructionsRetired =$
  $BranchInstructions +$
  $LoadInstructions +$
  $StoreInstructions +$
  $OtherInstructions +$
  $PackedSIMDComputationalInstructions$

- $CPI =$
  UNHALTED_CORE_CYCLES/
  INSTRUCTIONS_RETIRED

- $localPerformanceImprovement =$
  $CPI/EXPECTED\_CPI$

- $cyclesAfterImprovement =$
  UNHALTED_CORE_CYCLES/
  $localPerformanceImprovement$

- $totalCyclesAfterImprovement =$
  $totalCycles -$ **UNHALTED_CORE_CYCLES** $+$
  $cyclesAfterImprovement$

- $iMargin =$
  $100 - (totalCyclesAfterImprovement/totalCycles) * 100$

- $\%ofTotalCycles =$
  RS_UOPS_DISPATCHED CMASK=1 INV=1 $* 100/$
  UNHALTED_CORE_CYCLES

- $L2Miss\%ofcountedStalledCycles =$
  $L2MissImpact * 100/CountedStalledCycles$

- $L2Hit\%ofcountedStalledCycles =$
  $L2HitImpact * 100/CountedStalledCycles$

- $L1DTLBMiss\%ofcountedStalledCycles =$
  $L1DTLBMissImpact * 100/CountedStalledCycles$

- $LCPStalls\%ofcountedStalledCycles = LCPStallsImpact * 100/CountedStalledCycles$

- $StoreFwdStalls\%ofcountedStalledCycles = StoreFwdStallsImpact * 100/ CountedStalledCycles$

- $LoadsBlocked\%ofStoreFwdStallsCycles = LoadsBlockedbyUnknownAddressStoreImpact * 100/ StoreFwdStallsImpact$

- $LoadsOverlapped\%ofStoreFwdStallsCycles = LoadsOverlappedwithStoresImpact * 100/ StoreFwdStallsImpact$

- $LoadsSpanning\%ofStoreFwdStallsCycles = LoadsSpanningacrossCacheLinesImpact * 100/ StoreFwdStallsImpact$

- $Load\%ofallInstructions =$ **INST_RETIRED:LOADS** $* 100/ CountedInstructionsRetired$

- $Store\%ofallInstructions =$ **INST_RETIRED:STORES** $* 100/ CountedInstructionsRetired$

- $Branch\%ofallInstructions =$ **BRANCH_INSTRUCTIONS_RETIRED** $* 100/ CountedInstructionsRetired$

- $PackedSIMD\%ofallInstructions =$ **SIMD_COMP_INST_RETIRED:PACKED_SINGLE:PACKED_DOUBLE** $* 100/ CountedInstructionsRetired$

- $Other\%ofallInstructions = OtherInstructions * 100/ CountedInstructionsRetired$

6.2 Nehalem

Let us start with the definition of the following constants (specified in the Intel manuals):

- **I7_L1_DTLB_WALK_COMPLETED_CYCLES** = 35
- **I7_L1_ITLB_WALK_COMPLETED_CYCLES** = 35
- **I7_L2_HIT_CYCLES** = 6
- **I7_L3_UNSHARED_HIT_CYCLES** = 35
- **I7_OTHER_CORE_L2_HIT_CYCLES** = 60
- **I7_OTHER_CORE_L2_HITM_CYCLES** = 75
- **I7_L3_MISS_LOCAL_DRAM_HIT_CYCLES** = 225
- **I7_L3_MISS_REMOTE_DRAM_HIT_CYCLES** = 360
- **I7_L3_MISS_REMOTE_CACHE_HIT_CYCLES** = 180
- **I7_L3_MISS_OTHER_CYCLES** = 200
- **I7_IFETCH_L3_MISS_LOCAL_DRAM_HIT** = 200
- **I7_IFETCH_L3_MISS_REMOTE_DRAM_HIT** = 350
- **I7_IFETCH_L2_MISS_L3_HIT_NO_SNOOP** = 35
- **I7_IFETCH_L2_MISS_L3_HIT_SNOOP** = 60
- **I7_IFETCH_L2_MISS_L3_HITM** = 75
- **I7_IFETCH_L3_MISS_REMOTE_CACHE_FWD** = 180
- **EXPECTED_CPI** = 0.25

We monitor and we get the values of the following 41 microarchitectural events:

- **ARITH:CYCLES_DIV_BUSY**
- **BR_INST_EXEC:ANY**
- **BR_INST_EXEC:DIRECT_NEAR_CALL**
- **BR_INST_EXEC:INDIRECT_NEAR_CALL**
- **BR_INST_EXEC:INDIRECT_NON_CALL**
- **BR_INST_EXEC:NEAR_CALLS**
- **BR_INST_EXEC:NON_CALLS**
- **BR_INST_EXEC:RETURN_NEAR**
- **BR_INST_RETIRED:ALL_BRANCHES**
- **BR_INST_RETIRED:CONDITIONAL**
- **BR_INST_RETIRED:NEAR_CALL**
- **BR_MISP_EXEC:ANY**
- **CPU_CLK_UNHALTED:THREAD_P**
- **DTLB_LOAD_MISSES:WALK_COMPLETED**
- **INST_RETIRED:ANY_P**
- **ITLB_MISSES:WALK_COMPLETED**

- **L2_RQSTS:IFETCH_HIT**
- **L2_RQSTS:IFETCH_MISS**
- **MEM_INST_RETIRED:LOADS**
- **MEM_INST_RETIRED:STORES**
- **MEM_LOAD_RETIRED:L2_HIT**
- **MEM_LOAD_RETIRED:L3_MISS**
- **MEM_LOAD_RETIRED:L3_UNSHARED_HIT**
- **MEM_LOAD_RETIRED:OTHER_CORE_L2_HIT_HITM**
- **MEM_UNCORE_RETIRED:LOCAL_DRAM**
- **MEM_UNCORE_RETIRED:OTHER_CORE_L2_HITM**
- **MEM_UNCORE_RETIRED:REMOTE_CACHE_LOCAL_HOME_HIT**
- **MEM_UNCORE_RETIRED:REMOTE_DRAM**
- **OFFCORE_RESPONSE_0:DMND_IFETCH:LOCAL_DRAM**
- **OFFCORE_RESPONSE_0:DMND_IFETCH:OTHER_CORE_HITM**
- **OFFCORE_RESPONSE_0:DMND_IFETCH:OTHER_CORE_HIT_SNP**
- **OFFCORE_RESPONSE_0:DMND_IFETCH:REMOTE_CACHE_FWD**
- **OFFCORE_RESPONSE_0:DMND_IFETCH:REMOTE_DRAM**
- **OFFCORE_RESPONSE_0:DMND_IFETCH:UNCORE_HIT**
- **RESOURCE_STALLS:ANY**
- **SSEX_UOPS_RETIRED:PACKED_DOUBLE**
- **SSEX_UOPS_RETIRED:PACKED_SINGLE**
- **UOPS_DECODED:MS CMASK=1**
- **UOPS_ISSUED:ANY CMASK=1 INV=1**
- **ITLB_MISS_RETIRED**
- **UOPS_EXECUTED:0x3f**

Using these two sets of raw data and the following formulas, we can derive important and useful performance information:

- $TotalCycles =$
  CPU_CLK_UNHALTED:THREAD_P

- $L2HitImpact =$
  MEM_LOAD_RETIRED:L2_HIT $*$ **I7_L2_HIT_CYCLES**

- $L3UnsharedHitImpact =$
  MEM_LOAD_RETIRED:L3_UNSHARED_HIT $*$
  I7_L3_UNSHARED_HIT_CYCLES

- $L2OtherCoreHitImpact =$
  (**MEM_LOAD_RETIRED:OTHER_CORE_L2_HIT_HITM** $-$

MEM_UNCORE_RETIRED:OTHER_CORE_L2_HITM) ∗
I7_OTHER_CORE_L2_HIT_CYCLES

- $L2OtherCoreHitModifiedImpact =$
  MEM_UNCORE_RETIRED:OTHER_CORE_L2_HITM ∗
  I7_OTHER_CORE_L2_HITM_CYCLES

- $L3MissLocalDRAMHitImpact =$
  MEM_UNCORE_RETIRED:LOCAL_DRAM ∗
  I7_L3_MISS_LOCAL_DRAM_HIT_CYCLES

- $L3MissRemoteDRAMHitImpact =$
  MEM_UNCORE_RETIRED:REMOTE_DRAM ∗
  I7_L3_MISS_REMOTE_DRAM_HIT_CYCLES

- $L3MissRemoteCacheHitImpact =$
  MEM_UNCORE_RETIRED:REMOTE_CACHE_LOCAL_HOME_HIT ∗
  I7_L3_MISS_REMOTE_CACHE_HIT_CYCLES

- $L3MissOtherSourceImpact =$
  (**MEM_LOAD_RETIRED:L3_MISS** −
  $(L3MissLocalDRAMHitImpact +$
  $L3MissRemoteDRAMHitImpact +$
  $L3MissRemoteCacheHitImpact)) *$
  I7_L3_MISS_OTHER_CYCLES

- $L3MissTotalImpact =$
  $L3MissLocalDRAMHitImpact +$
  $L3MissRemoteDRAMHitImpact +$
  $L3MissRemoteCacheHitImpact +$
  $L3MissOtherSourceImpact$

- $L1DTLBMissImpact =$
  DTLB_LOAD_MISSES:WALK_COMPLETED ∗
  I7_L1_DTLB_WALK_COMPLETED_CYCLES

- $CountedStalledCyclesduetoLoadOps =$
  $L3MissTotalImpact +$
  $L2HitImpact +$
  $L1DTLBMissImpact +$
  $L3UnsharedHitImpact +$

$L2OtherCoreHitModifiedImpact +$
$L2OtherCoreHitImpact$

- $CyclesspentduringDIV\&SQRTOps =$
  ARITH:CYCLES_DIV_BUSY

- $TotalCountedStalledCycles =$
  $CountedStalledCyclesduetoLoadOps +$
  $CyclesspentduringDIV\&SQRTOps$

- $StalledCycles =$
  UOPS_EXECUTED:0x3f

- $\%ofTotalCycles =$
  $StalledCycles * 100/$
  CPU_CLK_UNHALTED:THREAD_P

- $L3Miss\%ofcountedStalledCycles =$
  $L3MissTotalImpact * 100/TotalCountedStalledCycles$

- $L2Hit\%ofcountedStalledCycles =$
  $L2HitImpact * 100/TotalCountedStalledCycles$

- $L1DTLBMiss\%ofcountedStalledCycles =$
  $L1DTLBMissImpact * 100/TotalCountedStalledCycles$

- $L3UnsharedHit\%ofcountedStalledCycles =$
  $L3UnsharedHitImpact * 100/$
  $TotalCountedStalledCycles$

- $L2OtherCoreHit\%ofcountedStalledCycles =$
  $L2OtherCoreHitImpact * 100/$
  $TotalCountedStalledCycles$

- $L2OtherCoreHitModified\%ofcountedStalledCycles =$
  $L2OtherCoreHitModifiedImpact * 100/$
  $TotalCountedStalledCycles$

- $DIV\&SQRTOps\%ofcountedStalledCycles =$
  $CyclesspentduringDIV\&SQRTOps * 100/$
  $TotalCountedStalledCycles$

- $CyclesIFETCHservedbyLocalDRAM =$
  OFFCORE_RESPONSE_0:DMND_IFETCH:LOCAL_DRAM $*$
  I7_IFETCH_L3_MISS_LOCAL_DRAM_HIT

- $CyclesIFETCHservedbyL3(Modified) =$
  OFFCORE_RESPONSE_0:DMND_IFETCH:OTHER_CORE_HITM $*$
  I7_IFETCH_L2_MISS_L3_HITM

- $CyclesIFETCHservedbyL3(CleanSnoop) =$
  OFFCORE_RESPONSE_0:DMND_IFETCH:OTHER_CORE_HIT_SNP $*$
  I7_IFETCH_L2_MISS_L3_HIT_SNOOP

- $CyclesIFETCHservedbyRemoteL2 =$
  OFFCORE_RESPONSE_0:DMND_IFETCH:REMOTE_CACHE_FWD $*$
  I7_IFETCH_L3_MISS_REMOTE_CACHE_FWD

- $CyclesIFETCHservedbyRemoteDRAM =$
  OFFCORE_RESPONSE_0:DMND_IFETCH:REMOTE_DRAM $*$
  I7_IFETCH_L3_MISS_REMOTE_DRAM_HIT

- $CyclesIFETCHservedbyL3(NoSnoop) =$
  OFFCORE_RESPONSE_0:DMND_IFETCH:UNCORE_HIT $*$
  I7_IFETCH_L2_MISS_L3_HIT_NO_SNOOP

- $TotalL2IFETCHmissImpact =$
  $CyclesIFETCHservedbyLocalDRAM +$
  $CyclesIFETCHservedbyL3(Modified) +$
  $CyclesIFETCHservedbyL3(CleanSnoop) +$
  $CyclesIFETCHservedbyRemoteL2 +$
  $CyclesIFETCHservedbyRemoteDRAM +$
  $CyclesIFETCHservedbyL3(NoSnoop)$

- $LocalDRAMIFECTHes\%Impact =$
  $CyclesIFETCHservedbyLocalDRAM * 100/$
  $TotalL2IFETCHmissImpact$

- $L3(Modified)IFECTHes\%Impact =$
  $CyclesIFETCHservedbyL3(Modified) * 100/$
  $TotalL2IFETCHmissImpact$

- $L3(CleanSnoop)IFECTHes\%Impact =$

$CyclesIFETCHservedbyL3(CleanSnoop) * 100/$
$TotalL2IFETCHmissImpact$

- $RemoteL2IFECTHes\%Impact =$
$CyclesIFETCHservedbyRemoteL2 * 100/$
$TotalL2IFETCHmissImpact$

- $RemoteDRAMIFECTHes\%Impact =$
$CyclesIFETCHservedbyRemoteDRAM * 100/$
$TotalL2IFETCHmissImpact$

- $L3(NoSnoop)IFECTHes\%Impact =$
$CyclesIFETCHservedbyL3(NoSnoop) * 100/$
$TotalL2IFETCHmissImpact$

- $TotalL2IFETCHmisses =$
L2_RQSTS:IFETCH_MISS

- $\%ofIFETCHesservedbyLocalDRAM =$
OFFCORE_RESPONSE_0:DMND_IFETCH:LOCAL_DRAM $* 100/$
L2_RQSTS:IFETCH_MISS

- $\%ofIFETCHesservedbyL3(Modified) =$
OFFCORE_RESPONSE_0:DMND_IFETCH:OTHER_CORE_HITM $* 100/$
L2_RQSTS:IFETCH_MISS

- $\%ofIFETCHesservedbyL3(CleanSnoop) =$
OFFCORE_RESPONSE_0:DMND_IFETCH:OTHER_CORE_HIT_SNP $* 100/$
L2_RQSTS:IFETCH_MISS

- $\%ofIFETCHesservedbyRemoteL2 =$
OFFCORE_RESPONSE_0:DMND_IFETCH:REMOTE_CACHE_FWD $* 100/$
L2_RQSTS:IFETCH_MISS

- $\%ofIFETCHesservedbyRemoteDRAM =$
OFFCORE_RESPONSE_0:DMND_IFETCH:REMOTE_DRAM $* 100/$
L2_RQSTS:IFETCH_MISS

- $\%ofIFETCHesservedbyL3(NoSnoop) =$
OFFCORE_RESPONSE_0:DMND_IFETCH:UNCORE_HIT $* 100/$
L2_RQSTS:IFETCH_MISS

- $\%ofL2IFETCHmisses =$
  L2_RQSTS:IFETCH_MISS $* 100/$
  (**L2_RQSTS:IFETCH_MISS** $+$
  L2_RQSTS:IFETCH_HIT)

- $L1ITLBMissImpact =$
  ITLB_MISSES:WALK_COMPLETED $*$
  I7_L1_ITLB_WALK_COMPLETED_CYCLES

- $TotalBranchInstructionsExecuted =$
  BR_INST_EXEC:ANY

- $\%ofMispredictedBranches =$
  BR_MISP_EXEC:ANY $* 100/$
  BR_INST_EXEC:ANY

- $DirectNearCalls\%ofTotalBranchesExecuted =$
  BR_INST_EXEC:DIRECT_NEAR_CALL $* 100/$
  $TotalBranchInstructionsExecuted$

- $IndirectNearCalls\%ofTotalBranchesExecuted =$
  BR_INST_EXEC:INDIRECT_NEAR_CALL $* 100/$
  $TotalBranchInstructionsExecuted$

- $IndirectNearNon-Calls\%ofTotalBranchesExecuted =$
  BR_INST_EXEC:INDIRECT_NON_CALL $* 100/$
  $TotalBranchInstructionsExecuted$

- $AllNearCalls\%ofTotalBranchesExecuted =$
  BR_INST_EXEC:NEAR_CALLS $* 100/$
  $TotalBranchInstructionsExecuted$

- $AllNonCalls\%ofTotalBranchesExecuted =$
  BR_INST_EXEC:NON_CALLS $* 100/$
  $TotalBranchInstructionsExecuted$

- $AllReturns\%ofTotalBranchesExecuted =$
  BR_INST_EXEC:RETURN_NEAR $* 100/$
  $TotalBranchInstructionsExecuted$

- $TotalBranchInstructionsRetired =$

BR_INST_RETIRED:ALL_BRANCHES

- $Conditionals\%ofTotalBranchesRetired =$
  BR_INST_RETIRED:CONDITIONAL $* 100/$
  $TotalBranchInstructionsRetired$

- $NearCalls\%ofTotalBranchesRetired =$
  BR_INST_RETIRED:NEAR_CALL $* 100/$
  $TotalBranchInstructionsRetired$

- $InstructionStarvation\%ofTotalCycles =$
  (UOPS_ISSUED:ANY CMASK=1 INV=1 $-$
  RESOURCE_STALLS:ANY) $* 100/$CPU_CLK_UNHALTED:THREAD_P

- $\%ofTotalCyclesspenthandlingFPexceptions =$
  UOPS_DECODED:MS CMASK=1 $* 100/$
  CPU_CLK_UNHALTED:THREAD_P

- $\#ofInstructionsperCall =$
  INST_RETIRED:ANY_P$/$
  BR_INST_EXEC:NEAR_CALLS

- $InstructionsRetired =$
  INST_RETIRED:ANY_P

- $ITLBMissRatein\% =$
  (ITLB_MISS_RETIRED$/$
  INST_RETIRED:ANY_P) $* 100$

- $BranchInstructions =$
  BR_INST_RETIRED:ALL_BRANCHES

- $LoadInstructions =$
  MEM_INST_RETIRED:LOADS

- $StoreInstructions =$
  MEM_INST_RETIRED:STORES

- $OtherInstructions =$
  $InstructionsRetired -$ MEM_INST_RETIRED:LOADS $-$
  MEM_INST_RETIRED:STORES $-$ BR_INST_RETIRED:ALL_BRANCHES

- $PackedUOPSRetired =$
  SSEX_UOPS_RETIRED:PACKED_DOUBLE $+$
  SSEX_UOPS_RETIRED:PACKED_SINGLE

- $CPI =$
  CPU_CLK_UNHALTED:THREAD_P/**INST_RETIRED:ANY_P**

- $localPerformanceImprovement =$
  $CPI/EXPECTED\_CPI$

- $cyclesAfterImprovement =$
  CPU_CLK_UNHALTED:THREAD_P/$localPerformanceImprovement$

- $totalCyclesAfterImprovement =$
  $totalCycles -$ **CPU_CLK_UNHALTED:THREAD_P** $+$
  $cyclesAfterImprovement$

- $iMargin =$
  $100 - (totalCyclesAfterImprovement/totalCycles) * 100$

- $Load\%ofallInstructions =$
  MEM_INST_RETIRED:LOADS $* 100/$
  INST_RETIRED:ANY_P

- $Store\%ofallInstructions =$
  MEM_INST_RETIRED:STORES $* 100/$
  INST_RETIRED:ANY_P

- $Branch\%ofallInstructions =$
  BR_INST_RETIRED:ALL_BRANCHES $* 100/$
  INST_RETIRED:ANY_P

- $Other\%ofallInstructions =$
  $OtherInstructions * 100/$
  INST_RETIRED:ANY_P

- $Packed\%ofallUOPSRetired =$
  $PackedUOPSRetired * 100/$
  UOPS_RETIRED:ANY

In Table 6.1 one can see the summary of the information calculated by the above formulas, and displayed in the module level analysis HTML result table.

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 EX.	% of cycles handling FP ex.
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
ITLB MISSES	L1 ITLB Miss Impact ITLB Miss Rate

DTLB MISSES	L1 DTLB Miss Impact L1 DTLB Miss % of Load Stalls
DIV & SQRT STALLS	Cycles executing DIV & SQRT Ops
L2 IFETCH MISSES	Total L2 IFETCH misses IFes served by Local DRAM IFes served by L3 (Modified) IFes served by L3 (Clean Snoop) IFes served by Remote L2 IFes served by Remote DRAM IFes served by L3 (No Snoop)
BRANCHES, CALLS & RETS	Total Branches Executed % of Mispredicted Branches Direct Near Calls Indirect Near Calls Indirect Near Non-Calls All Near Calls All Non Calls All Returns Conditionals
INSTRUCTION STATS	Branches, Loads, Stores, Packed

Table 6.1: *Types of micro-architectural events monitored*

Chapter 7

Conclusions

In this short report I introduced performance monitoring with hardware performance counters, and the 4-way analysis that we used to evaluate the performance of CMSSW and of its modules: overall, symbol, module level and modular symbol level analysis. We saw how Cycle Accounting Analysis, used in all our analysis approaches, gives a good insight on how a specific application performs by means of decomposition of cycles and most importantly of stalled cycles. We highlighted the need of using the 4 different approaches simultaneously in order to take advantage of their complementary informative capabilities. All the approaches rely directly or indirectly on *Perfmon2* interface to the PMU, and use both counting and sampling capabilities. This report was specific for CMSSW, but the same set of tools has been implemented for Gaudi and Geant4.

Bibliography

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*
http://www.intel.com/Assets/PDF/manual/253665.pdf

[2] *Intel 64 and IA-32 Architectures Optimization Reference Manual*
http://www.intel.com/Assets/PDF/manual/248966.pdf

[3] David Levinthal, *Cycle Accounting Analysis on Intel Core 2 Processors*
http://assets.devx.com/goparallel/18027.pdf

[4] David Levinthal, *Introduction to Performance Analysis on Intel Core 2 Duo Processors*
http://assets.devx.com/goparallel/17775.pdf

[5] David Levinthal, *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors*
http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

[6] Stephane Eranian, *Perfmon2: a standard performance monitoring interface for Linux*
http://perfmon2.sourceforge.net/perfmon2-20080124.pdf

[7] Stephane Eranian, *Perfmon2: a standard performance monitoring interface for Linux*
http://cscads.rice.edu/workshops/july2007/perf-slides-07/Eranian-Perfmon.pdf

[8] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, D. Ryabtsev, *Parallelization Made Easier with Intel Performance-Tuning Utility*
ftp://download.intel.com/technology/itj/2007/v11i4/2-parallelization/2-Parallelization_Made_Easier.pdf