National Technical University Of Athens
School Of Electrical And Computer Engineering

COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Improving Reliability & Efficiency Of Performance Monitoring In Linux

DIPLOMA THESIS

of

Maria N. Dimakopoulou

Supervisor: Nectarios Koziris
Professor N.T.U.A.

Athens, June 2014

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Improving Reliability & Efficiency Of Performance Monitoring In Linux

DIPLOMA THESIS

of

Maria N. Dimakopoulou

Supervisor: Nectarios Koziris
Professor N.T.U.A.

Approved by the committee on the 10th of June 2014.

....	....	....
Nectarios Koziris	Andreas Stafylopatis	Stéphane Eranian
Professor	Professor	Senior Software Engineer
N.T.U.A.	N.T.U.A.	Google

Athens, June 2014

....
Maria N. Dimakopoulou
Electrical & Computer Engineer

Abstract

Processor hardware performance counters have improved in quality and features in recent years. At the same time, the performance monitoring support in Linux has been significantly revamped with the development of the perf_events subsystem. Those factors concur in making performance monitoring a more common practice among developers. However, no performance analysis is possible without reliable hardware counter data.

In this thesis, we focus on a published correctness erratum in the performance monitoring unit of recent Intel processors when Hyper-Threading is enabled. This erratum causes cross hyper-thread hardware counter corruption and may produce unreliable results. We propose a cache-coherence style protocol that we implement in the Linux kernel to circumvent the issue by introducing cross hyper-thread dynamic event scheduling. We also introduce an event scheduling algorithm that achieves the optimal scheduling of events onto hardware counters at all times. The proposed optimizations do not require any user level changes and leverage the internal design of the perf_events subsystem. The source code has been contributed to the upstream Linux kernel.

Keywords
performance monitoring, hardware counters, PMU, hyper-threading, Linux kernel, perf_events, event scheduling

Περίληψη

Κατά τα τελευταία έτη, οι μετρητές επιδόσεων υλικού στους επεξεργαστές έχουν βελτιωθεί τόσο σε ποιότητα, όσο και σε χαρακτηριστικά. Ταυτόχρονα, η υποστήριξη παρακολούθησης επιδόσεων (performance monitoring) στο Linux έχει ανανεωθεί σημαντικά χάρη στην ανάπτυξη του υποσυστήματος perf_events. Αυτοί οι παράγοντες έχουν καταστήσει την παρακολούθηση επιδόσεων μια πιο κοινή πρακτική για τους προγραμματιστές. Ωστόσο, χωρίς αξιόπιστους μετρητές δεδομένων υλικού η ανάλυση επιδόσεων δεν είναι δυνατή.

Σε αυτή την εργασία, εστιάζουμε σε ένα δημοσιευμένο σφάλμα ορθότητας μετρήσεων στην μονάδα παρακολούθησης επιδόσεων (PMU) των πρόσφατων επεξεργαστών της Intel που συμβαίνει όταν η τεχνολογία Υπερ-Νηματισμού (Hyper-Threading) είναι ενεργοποιημένη. Αυτό το σφάλμα, μπορεί να προκαλέσει αλλοίωση των δεδομένων στους μετρητές υλικού μεταξύ των υπερ-νημάτων (hyper-threads), οδηγώντας έτσι σε αναξιόπιστα αποτελέσματα. Προκειμένου να παρακάμψουμε αυτό το πρόβλημα, προτείνουμε ένα πρωτόκολλο παρόμοιο με πρωτόκολλα συνάφειας μνημών cache (cache-coherence), το οποίο υλοποιούμε στον πυρήνα του Linux. Η λύση μας στηρίζεται στον προγραμματισμό των συμβάντων υλικού στους μετρητές επίδοσης κατά δυναμικό τρόπο, βάσει της κατάστασης των υπερ-νημάτων. Παρουσιάζουμε επίσης έναν αλγόριθμο που επιτυγχάνει πάντοτε βέλτιστο χρονοπρογραμματισμό των συμβάντων στους μετρητές υλικού. Οι βελτιστοποιήσεις που προτείνουμε δεν απαιτούν αλλαγές σε επίπεδο χρήστη και αξιοποιούν την εσωτερική σχεδίαση του υποσυστήματος perf_events. Ο πηγαίος κώδικας που αναπτύχθηκε έχει προσφερθεί στον πυρήνα του Linux.

Λέξεις-Κλειδιά
παρακολούθηση επίδοσεων, μετρητές υλικού, μονάδα παρακολούθησης επιδόσεων, υπερ-νηματισμός, πυρήνας του Linux, perf_events, χρονοπρογραμματισμός συμβάντων

Acknowledgments

While acknowledgments are often considered as a typical obligation of a thesis, I would like to go beyond and dedicate this work, from the depths of my heart, to all the people who have inspired me to accomplish little peaks one after the other and enabled me to look further into the horizon of life.

First of all, I would like to thank my professor, Nectarios Koziris, for introducing me to the field of Systems and Computer Architecture. It was through the quality of his lectures that I gained knowledge which would prove invaluable later, while working at Google. When during my second Google internship in Summer 2013, I was made aware of a serious hardware erratum corrupting performance measurements on Intel x86 processors, it was this knowledge of Computer Systems that gave me the insight into a solution to this problem, which had remained unsolved for the last three processor generations. I am also grateful to him for encouraging me to pursue this work and to lead it to some accomplishment I can be proud of.

I want to thank Kostis Nikas for his help, his advice and for the time he devoted to me during the writing of this thesis.

I would especially like to thank Stéphane Eranian for engaging me into the subject of Performance Monitoring and giving me the opportunity to learn about its utmost significance in software development and contribute in making it more reliable and powerful. With his constant attention and constructive guidance, he helped me move forward with this work, present it at CERN, contribute it in the Linux kernel and receive recognition from the Performance Monitoring community with an honorary award from Intel. Without him, this work and this thesis would not have existed.

I would also like to express my thanks and deep gratitude to my professor and mentor Andreas Stafylopatis for his encouragements to pursue the opportunity of an internship at Google in Summer 2012 and his substantial support as Dean of the ECE NTUA School at the time. His sincere care and advice have empowered me throughout these life-changing years of my studies and have contributed significantly to my personal evolution and my accomplishments up to this day.

Finally, this thesis is dedicated to my family and Stefanos. Without their continuous love and support simply nothing would be the same.

Contents

List of Figures

Chapter 1

Introduction

Nowadays, all processors provide a set of performance counters to measure many key micro-architectural events, such as the number of elapsed cycles, instructions executed, mispredicted branches, cache and TLB misses [1, 2, 3, 4]. In hardware the counters are implemented by a logical unit usually referred to as the Performance Monitoring Unit (PMU). PMUs are also present in other hardware devices beyond the processors, such as I/O, power, and memory controllers [5] where they can be used to measure memory bandwidth, cache coherency traffic, remote memory accesses, power consumption and read/write bandwidth to disk or network. PMUs can also be found in graphics cards [6].

It is possible to use the PMU to count occurrences of micro-architectural events or collect statistical profiles to determine where there may be resource bottlenecks using event-based sampling. The advantages of the PMU counters are, first, that they provide low-level data without requiring software modifications and second, that this information can be collected with very low overhead (usually $< 3\%$). The data they deliver, such as the number of cache misses, cannot be obtained by instrumenting software, but only by using cycle-accurate machine simulators. However, these are out of reach for many developers because they are usually reserved for internal use by hardware vendors and because they usually incur an extremely large slowdown at the execution of the workload.

The information collected by these micro-architectural events is crucial for any workload performance analysis. It is used for workload characterization, optimization of job placements with Google's CPI^2 [7] and provision of statistical data for compiler feedback directed optimizations with Google's AutoFDO [8]. It is also commonly used by profiling tools such as Intel's VTUNE [9] or Google's Gooda [10] to identify performance bottlenecks, such as instruction starvation or load latency and their location in the software. Based on the performance analysis, developers may be able to modify their programs to avoid certain bottlenecks, for instance, separate two fields of a structure into separate cache lines to avoid false sharing. Compilers may be able to optimize the hot path of a function to avoid branches. Information is also useful to the provision of machines based on workloads, for instance PMU data can tell whether a workload is CPU bound or not determining the choice of a processor model. Furthermore, each workload stresses certain parts of a micro-architecture differently. Knowing which micro-architectural element is the weakest link can help improve future processors to run workloads better, for instance by adding an extra load functional unit.

The Linux kernel provides full access to the hardware performance counters of hardware devices via the perf_events [11, 12] interface. This includes the processor, but also cache, PCIe, power, memory controllers. The interface can be used to count events or collect profiles on a per-thread or per-CPU basis from small handheld devices, such as phones or tablets, to large servers. It supports all the major processor architectures from Intel x86 and IBM Power to the various flavors of ARM-based chips. It can also handle many non-core PMUs found on server processors, such as Intel Xeons. All Linux distributions comes with an open-source tool called Perf which can exercise all aspects of the interface across all the processor architectures.

With a powerful and integrated kernel monitoring infrastructure, more developers are inclined to use hardware counters to analyze their applications. New usage models are emerging with live automatic feedback loop system such as Google's CPI^2 [7]. As more people, most of whom are not necessarily processor micro-architecture experts, come to rely on counter data, it is very important to ensure the correctness of the measurements they produce and minimize the overhead of monitoring.

Understanding low-level raw performance data is not an easy task given the complexity of today's processors. Tools can help abstract some of that complexity by using higher level metrics but that is possible only if the low-level data is trustworthy. The kernel interface providing access to the PMU must be stable and thoroughly tested. At the hardware level, micro-architectural events must be validated to ensure they count what they are supposed to at all times. Validating events can be challenging, as it entails developing subtle micro-benchmarks with known behaviors and verifying that the event counts make sense. The complexity of this job is too often underestimated as the PMU is rarely considered a critical component of a hardware device, e.g., a processor can operate perfectly fine with a PMU that produces invalid counts for cache misses. If the PMU is known to be unreliable and expert-only, users will turn away and it will not be further developed, leaving potential performance gain opportunities unexploited.

Recent Intel processors with Hyper-Threading [13] have a published erratum which seriously impacts the correctness of hardware counters under certain conditions, potentially leading to very large counter corruptions. Currently, there is no hardware or firmware fix available for the impacted processors. In this thesis, we describe a cache-coherence style protocol that we have implemented in the Linux kernel to completely eliminate the corruption without any changes to user tools or metrics. Our software solution to this erratum leverages the design of the perf_events subsystem and in particular the way it controls how events are programmed onto counters, i.e., scheduled onto the PMU counters.

The event scheduling algorithm is at the core of the subsystem. Not all events can be measured on all the counters due to hardware constraints. The goal of the event scheduling algorithm is to assign events to valid counters while at the same time try to maximize the use of the counters. If events are programmed on the wrong counters, they may silently count incorrectly. Thus, the scheduling algorithm is critical to ensure the correctness of the performance monitoring data.

When there are more events to measure than counters or when there are events competing for the same counters, the perf_events subsystem can time-share the counters. In doing so, it provides flexibility for monitoring tools at the cost of accuracy. The total count of a multiplexed event is obtained by scaling with a timing factor the raw count accumulated each time the event is scheduled. This approach works well, when the rate of occurrence of the event is constant but it is not so accurate if the monitored workload has rapidly changing phases. In order to mitigate this effect and improve accuracy, the scheduling algorithm needs to program as many events as possible on the available counters. Maximizing counter usage also minimizes the overhead of monitoring because event do not need to reprogrammed as frequently.

The current perf_events scheduling algorithm uses a first match greedy approach which works well when events are mostly unconstrained, whereas it is not so efficient when many events are constrained. A consequence of our erratum workaround is that events which do not have hardware constraints, become constrained based on what is scheduled on the processor's hyper-threads. More constraints put more pressure on the existing scheduling algorithm and this results in degraded quality of produced schedules. In this thesis, following our work on the erratum, we describe how, based on an advanced graph algorithm, we have improved the perf_events event scheduler with an optimal scheduling algorithm.

Our thesis is decomposed in three majors parts. In the first part, we describe the performance monitoring hardware of recent Intel processors and give an overview of the Linux perf_events subsystem focusing on the event scheduling algorithm. In the second part, we describe the hardware erratum and give examples of the measurement corruption it incurs. Then, we enumerate the possible solutions and justify why the our workaround is by far the most preferable approach. We describe our solution and we demonstrate our results. In the third part, we analyze the current event scheduling algorithm and explain how we have identified a better approach based on a graph algorithm. We describe our implementation and evaluate our solution.

The work presented in our thesis will be integrated to the upstream Linux kernel.

Chapter 2

Performance Monitoring

2.1 PMU hardware

Every modern processor provides a set of hardware counters to measure micro-architectural events, such as the number of elapsed clock cycles, instructions retired and cache misses [1, 2, 3]. Those counters are implemented in silicon by a logical unit called the Performance Monitoring Unit (PMU). Nowadays, PMUs are found in processor physical cores, last level cache controllers, memory controllers, graphics cards and I/O devices [5, 14, 6]. They provide crucial data to understand how the hardware resources are used by software. Interpretation of the data can identify source of bottlenecks and give hints on how to eliminate them.

In our thesis, we focus on Intel Sandy Bridge, Ivy Bridge and Haswell processors. They all have a very sophisticated and powerful PMU [1]. When Hyper-Threading is enabled, each logical CPU has 3 fixed counters and 4 generic counters. The fixed counters measure only one event each, whereas the generic counters can be programmed to measure up to 4 different events simultaneously. The counters are implemented by privileged model-specific registers (MSR). There is a configuration register where the event is programmed and a counter register where the occurrences are accumulated. The width of the counter can vary. It is 48-bit on the processors we use in our thesis. There is also a set of global control and status registers to start and stop the PMU easily. Managing the PMU requires kernel-level support, either in the form of a device driver or a system call.

The list of supported events is specific to each processor implementation as it is closely tied to the micro-architecture [1]. However, each implementation tends to build on the previous. The counters can be programmed to count occurrences of an event or to collect a profile using event-based sampling. Counters can interrupt on overflow, which is how sampling is implemented. To capture a sample after p occurrences of an event, a counter is programmed to the value of $-p$. When the counter overflows, i.e., wraps back to 0, an interrupt is generated. The kernel catches the interrupt and saves the current instruction pointer in a sampling buffer which is eventually parsed by a performance tool.

Events may have counter constraints due to hardware limitations. For instance, some events may only be measured on a specific PMU counter only, e.g, counter 2, otherwise incorrect counts may be captured. Some events may require an extra configuration register and therefore only one instance of the event may be measured at any time. Any kernel driver or tool needs to enforce these restrictions.

2.2 Operating systems infrastructure

2.2.1 Linux perf_events interface

Since Linux kernel version 2.6.31, there is an official kernel interface to access the hardware performance counters. It is called perf_eventsand it provides a high-level, generic interface to count and sample hardware and software events or Linux kernel trace-points. The architecture of this kernel subsystem is depicted in Fig. 2.1. The user visible interface provides a new system call, perf_event_open(), and a series of new file entries in sysfs to simplify event naming and configuration for tools. The core logic is encapsulated into the generic layer, common to all processor architectures. There is a layer per architecture and a set of PMU specific support routines to handle model specific features. To make it easy to develop tools across various hardware platforms, and unlike many other interfaces such as OProfile [15], the interface is event-driven. Users pass events to measure and not register value pairs. The kernel is responsible for programming these events onto the correct counters, i.e., managing the PMU resource. Users are never aware of the actual number of counters nor of event constraints.

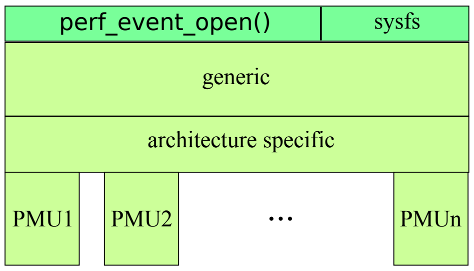


Figure 2.1: Kernel architecture of perf_events subsystem

The interface provides a set of generic events for basic monitoring, such as cycles, instructions and branches. These are mapped onto actual events by the kernel. It is also possible to program any model-specific event supported by the host PMU. To program an event, a tool uses the new perf_event_open() system call. Each event is then identified by a file descriptor. To start or stop an event, the file descriptor is passed to the standard ioctl() system call with a specific command, such as PERF_IOC_ENABLE to activate an event. To read an event, the file descriptor is passed to the regular read() system call.

Each event is managed individually. It is possible to create event groups to ensure a set of events is always measured together, which helps with certain metric computations. An event can be measured in system-wide or per-thread mode, where it is attached to a physical core or a logical CPU respectively. On a machine with 8 logical CPUs, it is necessary to create 8 instances of an event, each attached to one logical CPU. Similarly to monitor a multi-threaded program, there needs to be one instance of each event attached to each thread. As of Linux kernel 3.14, the perf_events subsystem supports all major processors on which Linux runs, from mainframes to hand-held devices.

2.2.2 Linux OProfile interface

The OProfile interface [15] is a Linux specific hardware performance monitoring interface inspired by DEC's DCPI [16]. It provides access to the processor hardware counters. For a long time, it has been the official monitoring interface of the Linux kernel providing only system-wide profiling capabilities across all major processor architectures. Nowadays it has been superseded by perf_events. The whole infrastructure consists of a kernel level driver, a user level daemon (oprofiled), and a set of commands to start and stop monitoring and process the samples: opcontrol, opreport, opannotate. These commands interact with the daemon which is responsible for communicating with the kernel and for symbolizing the samples, i.e., associate symbols to sampled addresses.

Although OProfile is deprecated, the user level commands persist and are now implemented on top of the perf_events interface to maintain backward compatibility for the many scripts developed for the OProfile command set.

The OProfile kernel interface is a register-driven interface. The user level code is passing (register, value) pairs to program an event on a counter.

2.2.3 Non-Linux interfaces

Hardware performance monitoring interfaces exist in many other open-source or commercial operating systems.

The FreeBSD [17] operating system provides a system-call based interface called hwpmc [18]. It exposes a counter based interface and supports counting and sampling on a per-process or system-wide basis. The user level tool is called pmcstat and interacts with the kernel via a helper library called libpmc.

Commercial operating systems such as HPUX, Oracle's Solaris and IBM AIX also have hardware performance monitoring interfaces. However, they are not public and they are used by proprietary tools such as Oracle Solaris Studio Performance Analyzer [19].

For Microsoft Windows, there is no standard kernel interface. Instead, tools come with their own drivers. On Intel, the VTUNE amplifier XE analysis[9] tool comes with an open-source driver called sep. The same driver is also used by Intel's Performance Bottleneck Analyzer [20, 21]. The driver provides a register based interface to program the hardware performance counters.

2.3 Performance monitoring tools

2.3.1 Perf

The perf tool [11, 12] is the Linux official open-source performance monitoring tool. It is developed as part of the Linux kernel and is offered by all standard Linux distributions.

This is a command line tool used to collect performance data from many different counter sources such as hardware counters, kernel software counters and trace-points. From each source, it is possible to count event occurrences or collect event based statistical profiles. It is possible to measure on system-wide or per-thread mode.

The tool is built on top of the Linux kernel perf_events interface and offer access to all the features of that interface.

For profiling, the tool operates in a two-stage process. The profile is collected using the perf record command. The function level profile is obtained with perf report. The assembly and source level profile is generated by the perf annotate command. There is a simple text-based user interface but no advanced cycle analysis. Below we demonstrate a simple example of profiling the `dd` command:

```
$ perf record dd if=/dev/urandom of=/dev/null count=100000

$ perf report --stdio
# Samples: 12K of event 'cycles'
# Event count (approx.): 10659132347
#
# Overhead  Command      Shared Object                  Symbol
# ........  .......  .................  ......................
#
    57.99%       dd  [kernel.kallsyms]  [k] sha_transform
    18.55%       dd  [kernel.kallsyms]  [k] _mix_pool_bytes
    16.50%       dd  [kernel.kallsyms]  [k] extract_buf
     1.50%       dd  [kernel.kallsyms]  [k] __ticket_spin_lock
```

For counting, the perf stat must be used. It can aggregate counts per process, per core and per processor socket. It is also possible to print count deltas at regular time intervals:

```
$ perf stat dd if=/dev/urandom of=/dev/null count=100000
100000+0 records in
100000+0 records out
```

```
51200000 bytes (51 MB) copied, 3,24949 s, 15,8 MB/s
 Performance counter stats for 'dd if=/dev/urandom of=/dev/null count=100000':
       3251,489215 task-clock (msec)       #    1,000 CPUs utilized
               284 context-switches        #    0,087 K/sec
                 3 cpu-migrations          #    0,001 K/sec
               252 page-faults             #    0,078 K/sec
    10 643 745 482 cycles                  #    3,273 GHz
     3 399 485 751 stalled-cycles-frontend #   31,94% frontend cycles idle
   <not supported> stalled-cycles-backend
    25 823 470 801 instructions           #    2,43  insns per cycle
                                          #    0,13  stalled cycles per insn
       459 733 444 branches               #  141,392 M/sec
           125 222 branch-misses           #    0,03% of all branches

       3,251198688 seconds time elapsed
```

The list of supported events depends on the underlying hardware platform. However, the perf_events subsystem defines a set of generic events which the tool can directly leverage as shown in the example above, e.g., cycles, instructions. These events are mapped by the kernel onto actual hardware events if they exist.

The kernel may also export model specific events in the sysfs filesystem. They can be used directly by the perf stat tool. This is demonstrated in the example below where the tool is used to access a processor socket level set of counters called RAPL which measures the energy consumption of the chip. The count deltas are printed every second for 100s. Events may have units which are also shown: here the processor in consuming about 2.10 Joules per second, i.e, Watts. as follows:

```
$ perf stat -a -e power/energy-cores/,power/energy-pkg/,power/energy-gpu/\
  -I 1000 sleep 100
#          time             counts   unit events
     1.000123322              2.11 Joules power/energy-cores/       [100.00%]
     1.000123322              5.96 Joules power/energy-pkg/         [100.00%]
     1.000123322              0.31 Joules power/energy-gpu/
     2.000354464              2.09 Joules power/energy-cores/
     2.000354464              5.95 Joules power/energy-pkg/
     2.000354464              0.31 Joules power/energy-gpu/
     ...
```

2.3.2 Gooda

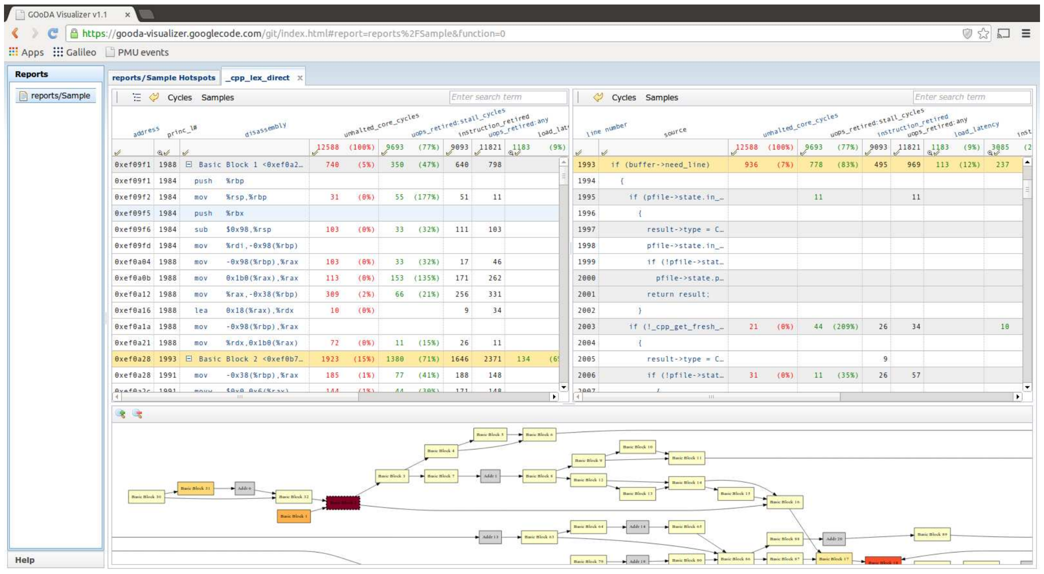
The Gooda [10] tool is an open-source performance analysis tool developed by Google for Linux.

It provides a system-wide cycle-breakdown analysis using the hardware counters of Intel processors. It breaks down how each cycle is spent, i.e., whether it does useful or useless work. Stalled cycles are classified in high level categories, such as load latency or instruction starvation. These get eventually mapped onto actual hardware events.

The tool is built on top of the Linux perf_events subsystem and the Perf tool. To collect the system-wide profile, Gooda uses the perf record command. The profile data is then analyzed by Gooda to produce a series of text files (JSON format) which contain the full analysis. Those files can then be visualized in a standard web browser using a Javascript program.

The web-based GUI allows navigating from the process level analysis down to the basic-block level analysis providing assembly, control flow graph and source views.

The advantage of this web-based tool is that the entire analysis is contained in the produced text files. The analysis can be shared easily by simply passing URLs. Remote users do not need the binaries or the source code of the monitored programs to look at the data.



address	princ_l#	disassembly	unhalted_core_cycles		uops_retired:stall_cycles		instruction_retired	uops_retired:any	load_latency	
			12588	(100%)	9693	(77%)	9093	11821	1183	(9%)
0xef09f1	1988	⊟ Basic Block 1 <0xef0a2…	740	(5%)	350	(47%)	640	798		
0xef09f1	1984	push %rbp								
0xef09f2	1984	mov %rsp,%rbp	31	(0%)	55	(177%)	51	11		
0xef09f5	1984	push %rbx								
0xef09f6	1984	sub $0x98,%rsp	103	(0%)	33	(32%)	111	103		
0xef09fd	1984	mov %rdi,-0x98(%rbp)								
0xef0a04	1988	mov -0x98(%rbp),%rax	103	(0%)	33	(32%)	17	46		
0xef0a0b	1988	mov 0x1b0(%rax),%rax	113	(0%)	153	(135%)	171	262		
0xef0a12	1988	mov %rax,-0x38(%rbp)	309	(2%)	66	(21%)	256	331		
0xef0a16	1988	lea 0x18(%rax),%rdx	10	(0%)			9	34		
0xef0a1a	1988	mov -0x98(%rbp),%rax								
0xef0a21	1988	mov %rdx,0x1b0(%rax)	72	(0%)	11	(15%)	26	11		
0xef0a28	1993	⊟ Basic Block 2 <0xef0b7…	1923	(15%)	1380	(71%)	1646	2371	134	(6
0xef0a28	1991	mov -0x38(%rbp),%rax	185	(1%)	77	(41%)	188	148		
0xef0a2c	1991	movw $0x0,0x6(%rax)	144	(1%)	44	(30%)	171	148		

line number	source	unhalted_core_cycles		uops_retired:stall_cycles		instruction_retired	uops_retired:any	load_latency		inst
		12588	(100%)	9693	(77%)	9093	11821	1183	(9%)	3085 (2
1993	if (buffer->need_line)	936	(7%)	778	(83%)	495	969	113	(12%)	237
1994	{									
1995	if (pfile->state.in_…			11			11			
1996	{									
1997	result->type = C…									
1998	pfile->state.in_…									
1999	if (!pfile->stat…									
2000	pfile->state.p…									
2001	return result;									
2002	}									
2003	if (!_cpp_get_fresh_…	21	(0%)	44	(209%)	26	34			10
2004	{									
2005	result->type = C…					9				
2006	if (!pfile->stat…	31	(0%)	11	(35%)	26	57			
2007	{									

Figure 2.2: The Gooda analysis interface

2.3.3 Intel VTUNE Amplifier XE

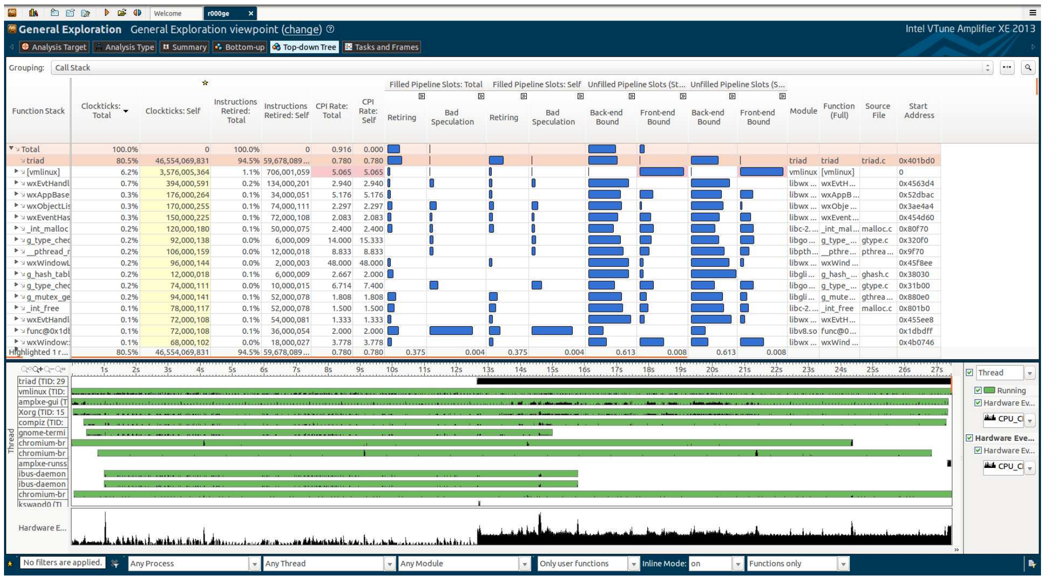
Intel VTUNE Amplifier XE [9] is an advanced commercial tool available on Windows and Linux. It works on a variety of Intel hardware platforms: laptops, desktops, servers and co-processors such as Xeon Phi. It leverages the hardware performance counters on those platforms to provide a set of system-wide analysis, such as memory bandwidth, top-down cycle analysis [22, 23]. It is possible to drill down from processes, to functions and assembly.

On Linux, it is composed of three parts:

1. amplxe-gui: the graphical user interface (GUI)
2. amplxe-cl: the command line tool (sepcli) to actually collect the data
3. sep: the open-source Linux kernel driver

On Linux, the tool does not use the official perf_events interface but instead the sep open-source driver for compatibility with Windows systems.

The rich GUI interface allows many filtering and navigation options. Fig. 2.3 shows a screenshot of a top-down analysis of a simple test program, called triad which streams data from and to memory. As expected, the analysis shows the program is back-end bound, i.e., it is waiting for memory accesses.



Grouping: Call Stack

Function Stack	Clockticks: Total	Clockticks: Self	Instructions Retired: Total	Instructions Retired: Self	CPI Rate: Total	CPI Rate: Self	Filled Pipeline Slots: Total: Retiring	Filled Pipeline Slots: Total: Bad Speculation	Filled Pipeline Slots: Self: Retiring	Filled Pipeline Slots: Self: Bad Speculation	Unfilled Pipeline Slots (St...: Back-end Bound	Unfilled Pipeline Slots (St...: Front-end Bound	Unfilled Pipeline Slots (S...: Back-end Bound	Unfilled Pipeline Slots (S...: Front-end Bound	Module	Function (Full)	Source File	Start Address
Total	100.0%	0	100.0%	0	0.916	0.000												
triad	80.5%	46,554,069,831	94.5%	59,678,089…	0.780	0.780									triad	triad	triad.c	0x401bd0
[vmlinux]	6.2%	3,576,005,364	1.1%	706,001,059	5.065	5.065									vmlinux	[vmlinux]		0
wxEvtHandl	0.7%	394,000,591	0.2%	134,000,201	2.940	2.940									libwx …	wxEvtH…		0x4563d4
wxAppBase	0.3%	176,000,264	0.1%	34,000,051	5.176	5.176									libwx …	wxAppB…		0x52dbac
wxObjectLis	0.3%	170,000,255	0.1%	74,000,111	2.297	2.297									libwx …	wxObje…		0x3ae4a4
wxEventHas	0.3%	150,000,225	0.1%	72,000,108	2.083	2.083									libwx …	wxEvent…		0x454d60
_int_malloc	0.2%	120,000,180	0.1%	50,000,075	2.400	2.400									libc-2.…	_int_mal…	malloc.c	0x80f70
g_type_chec	0.2%	92,000,138	0.0%	6,000,009	14.000	15.333									libgo…	g_type_…	gtype.c	0x320f0
__pthread_r	0.2%	106,000,159	0.0%	12,000,018	8.833	8.833									libpth…	__pthre…	pthrea…	0x9f70
wxWindowL	0.2%	96,000,144	0.0%	2,000,003	48.000	48.000									libwx …	wxWind…		0x45f8ee
g_hash_tabl	0.2%	12,000,018	0.1%	6,000,009	2.667	2.000									libgli…	g_hash_…	ghash.c	0x38030
g_type_chec	0.2%	74,000,111	0.0%	10,000,015	6.714	7.400									libgo…	g_type_…	gtype.c	0x31b00
g_mutex_ge	0.2%	94,000,141	0.1%	52,000,078	1.808	1.808									libgli…	g_mute…	gthrea…	0x880e0
_int_free	0.1%	78,000,117	0.1%	52,000,078	1.500	1.500									libc-2.…	_int_free	malloc.c	0x801b0
wxEvtHandl	0.1%	72,000,108	0.1%	54,000,081	1.333	1.333									libwx …	wxEvtH…		0x455ee8
func@0x1dl	0.1%	72,000,108	0.1%	36,000,054	2.000	2.000									libv8.so	func@0…		0x1dbdff
wxWindow::	0.1%	68,000,102	0.0%	18,000,027	3.778	3.778									libwx …	wxWind…		0x4b0746
Highlighted 1 r…	80.5%	46,554,069,831	94.5%	59,678,089…	0.780	0.780	0.375	0.004	0.375	0.004	0.613	0.008	0.613	0.008				

No filters are applied. | Any Process | Any Thread | Any Module | Only user functions | Inline Mode: on | Functions only

Figure 2.3: VTUNE amplifier top-down analysis screenshot

2.3.4 Intel Performance Bottleneck Analyzer

The Intel Performance Bottleneck Analyzer framework (PBA) [21, 20] is an experimental monitoring tool which uses a different approach to analyze performance. It is built on top of the same kernel driver interface as VTUNE, namely the sep driver. The tool utilizes the PMU hardware of Intel X86 processors, and in particular the ability to sample taken branches. It uses these performance monitoring data to recreate the hottest paths of instruction execution through a binary in order to find bottlenecks along it. It also samples common stalls events. The recreated paths of execution are then passed through an analysis related to well known code generation issues. The flow of analysis for this tool is illustrated in Fig. 2.4.

The execution paths are displayed in a graph with addresses on the horizontal axis and events histograms on the vertical axis in Fig. 2.5.

Any spike denotes a high event count, i.e., a potential bottleneck cause which users can further analyze.

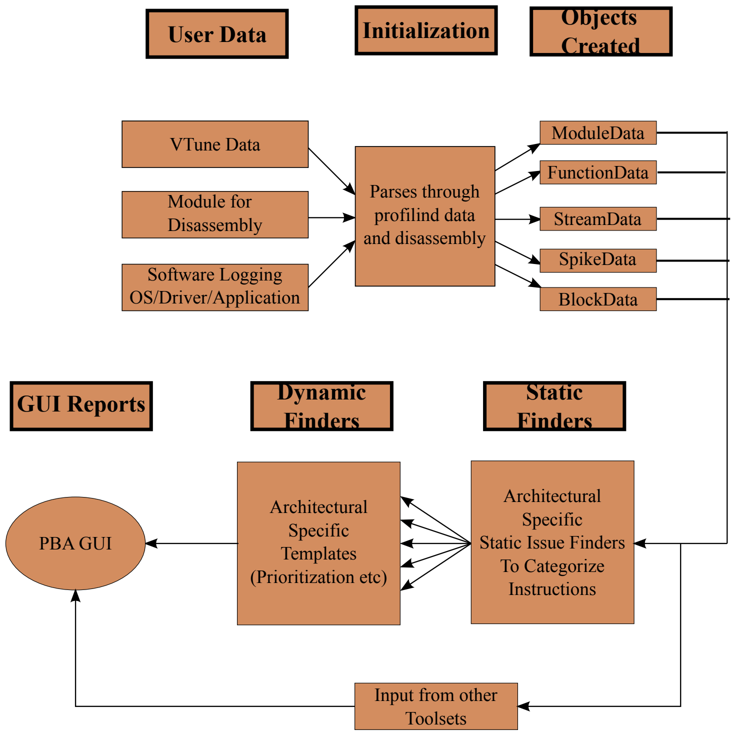

User Data
Initialization
Objects Created
VTune Data
Module for Disassembly
Software Logging OS/Driver/Application
Parses through profilind data and disassembly
ModuleData
FunctionData
StreamData
SpikeData
BlockData
GUI Reports
Dynamic Finders
Static Finders
PBA GUI
Architectural Specific Templates (Prioritization etc)
Architectural Specific Static Issue Finders To Categorize Instructions
Input from other Toolsets

Figure 2.4: Intel PBA Flow of Analysis

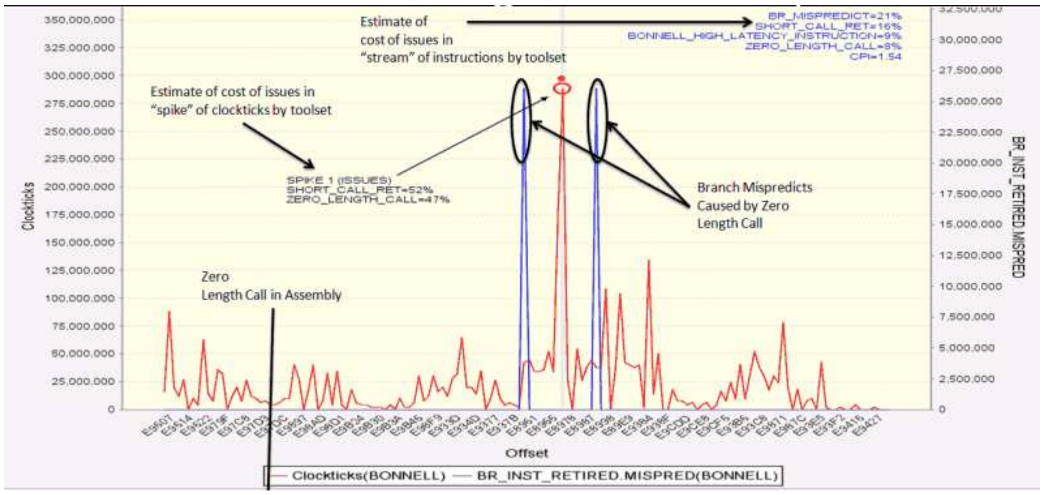


Figure 2.5: PBA Relating Static and Dynamic Data

Chapter 3

PMU Event Scheduling

3.1 Generic layer

Users can measure an arbitrary long list of events. Multiple tools may monitor the same process or processor in parallel. To ensure correct measurements, the kernel, which is responsible for managing the PMU resource, must arbitrate counter usage and assign events to the proper counters. This is called *event scheduling*. It takes as input a list of events and the output is an assignment of these events to the hardware counters.

Scheduling occurs when events are added or removed and on context switches for per-thread events. In case the PMU is over-subscribed, i.e., there are more events than counters, the kernel can time multiplex events onto the counters. Each time the multiplexing timer expires, the current events are scheduled out and they get replaced by others.

Multiplexing may also occur as a consequence of event scheduling conflicts, i.e., two or more events competing for the same counter. The event scheduler should attempt to maximize counter usage in order to minimize the need for time-sharing which could incur inaccuracies.

Each PMU may have different scheduling restrictions. Therefore, the actual scheduling algorithm is implemented in the architecture specific layer of the Linux kernel. If multiple PMUs share the same kind of restrictions, they can use the same scheduling algorithm. This is the case for all Intel x86 core PMUs which we describe hereafter.

Scheduling occurs independently on each CPU and operates at the event group granularity. A group is treated atomically. Either all the events in a group can be scheduled or none is scheduled. We assume one event per group in our description. In the generic perf_events layer (c.f. Fig. 2.1), events are inserted into one of two lists based on their type: per-thread or system-wide. To keep the description simple, we assume one event list.

Scheduling is always driven from the generic perf_events layer. Events are incrementally passed down from the linked list in the generic layer to the low level event scheduling algorithm. Scheduling operates in passes, P_x, using an incremental event window on the linked list. In the first pass (P_1), the window starts with a size of 1, i.e., one event is passed down. If it can be scheduled, then, in the second pass (P_2), the window grows to a size of two, i.e., first and second events are passed down, and so on and so forth. The passes stop at the first scheduling error or when the window contains all the events on the linked list. In case of an error with a window size of K, the counter assignment generated for size $K-1$ is programmed.

The algorithm is bound by the number of counters. If the PMU has N counters, scheduling stops when at most N events are passed down (window size $K \leq N$). This guarantees that if an arbitrary long list of events is provided, the system will not slow down proportionally due to time consuming event scheduling.

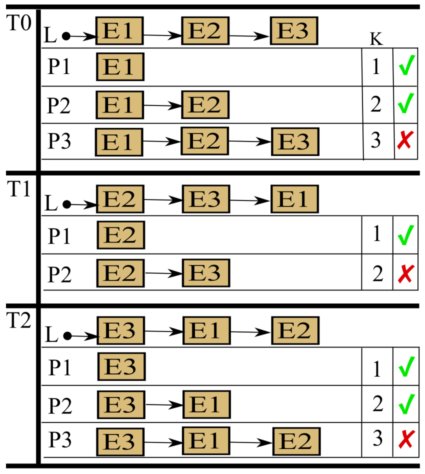


Figure 3.1: Event list scheduling and rotation example

In Fig. 3.1, we illustrate the iterative process between the generic and architecture specific layers with 3 events: E1, E2, E3. The events E2 and E3 are conflicting, i.e., E2 and E3 can only be measured on the same specific PMU counter and thus, they cannot be scheduled simultaneously. Fig. 3.1 shows 3 successive iterations of the algorithm: $T_0 - T_2$. In the first iteration of the algorithm (T_0), the first two passes (P_1, P_2) succeed but the third (P_3) fails because there is no counter available for E3, as it is already occupied by E2. Thus, only two events are scheduled during this iteration, these from the P_2 pass. Once scheduling is complete, events are actually programmed onto the counters and activated. On the next scheduling iteration (T_1), the list of events is rotated, i.e., the head is moved to the tail and the scheduling algorithm starts again. On the second iteration, only one event is scheduled from the P_1 pass. Eventually, on the third scheduling iteration (T_2), E3 gets to the head of the list and is scheduled in P_1 and P_2 passes. Hence, the algorithm guarantees that all events are eventually scheduled.

3.2 Intel x86 core PMU scheduling algorithm

The current Intel x86 algorithm uses a greedy, first match approach to assign events to counters. At each pass, once an event is assigned to a counter, it cannot be reassigned even though it could run on another counter. Events may have static constraints, i.e., they may run on a limited subset of counters. For each Intel x86 PMU, the kernel maintains a table of constrained events keyed off of event codes. For each constrained event, a bit mask of supported counters is returned to the scheduling algorithm. Generic hardware counters are indexed starting at 0 and thus each bit in the mask represents a supported counter. For instance, a mask of 0x3 means counters 0 and 1 are supported. We define the *weight* of a constraint as the number of set bits. The bigger the weight, the less constrained an event is, i.e., more counter choices.

```
struct event_constraint snb_constraints[]={
  CNST(0x48,0x4),    /*L1D_PEND_MISS.PENDING */
  U_CNST(0x01c0,0x2),/*INST_RETIRED.PREC_DIST*/
};
```

In the code snippet above, we show an excerpt of the constraint table for the Intel SandyBridge processor. Event L1D_PEND_MISS.PENDING, with code 0x48, can only be programmed on counter 2. If an event is not defined in the table, it can run on any generic counters and therefore in a PMU architecture with 4 generic counters, the constraint mask is 0xf.

Once the event list of window size K is passed by the generic layer to the low-level Intel x86 scheduler proceeds in the following 2 steps:

1. The event constraints are collected from the constraint tables and the weight of each event is calculated. The K events are distributed to the different weight categories.

2. The scheduling algorithm is invoked and tries to assign the K events to counters starting from smallest weight category (most constrained events) and moving to biggest one (least constrained events). For each event in a weight category, the algorithm iterates over the N PMU counters until the first counter matching this event's constraints is found.

```
sort weights[] in ascending order;
for each weight in weights[]
 for each (event in weight.events[])
  for each (counter in counters[])
   if ((event.constraint allows counter) and (counter is available)) {
    assign event to counter;
    mark counter as unavailable;
    break;
   }
```

The event is assigned to this counter without a possibility of future reassignment to another matching counter. Therefore, the scheduling is done based on a greedy, first match approach algorithm. This step is described at the pseudocode below.

As explained before, the window size K is bound by the number of PMU counters N. Thus, the complexity of the Intel x86 scheduling algorithm is $O(N^2)$.

An assignment example for 3 events is given in Fig. 3.2. The event window grows from 1 to 3. At each pass $(P_1 - P_3)$, the events are scheduled in constrained order. With a window size of 2 (E1, E2), E1 is scheduled first and second E2, because of their weights, respectively 1 and 4. With a window size of 3 (E1, E2, E3), E1 is scheduled first, second comes E3 and last E2. For each pass, the array on the right of the figure shows the counter assignment.

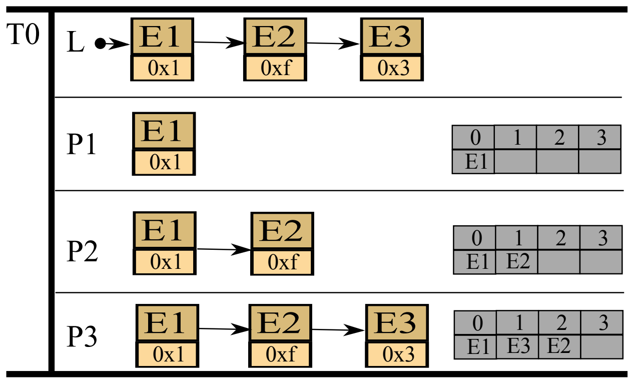


Figure 3.2: Event scheduling on x86 PMU architecture

To minimize the cost of scheduling, for each event, the algorithm first tries to reuse the counter assigned to the event the previous time, i.e., fast path scheduling. If that works, then nothing else is needed. If this fails, then the normal algorithm (normal path) is executed.

3.3 Handling of failures

When no assignment is possible for a window size of K, an error is returned to the generic layer which stops trying to increase the event window size. The previous window of $K - 1$ events is scheduled on the counters.

To ensure all events get a chance to be scheduled, errors trigger multiplexing. When the multiplexing timer expires (default timeout is each timer tick), the linked list is rotated by one event and a new scheduling iteration is performed starting with a window size of 1. All common events are guaranteed to be scheduled because they all eventually reach the head of the linked list and thus will be scheduled, at the worst case, with the event window size of 1.

Chapter 4

PMU Hardware Erratum

4.1 Problem description

On Intel Sandy Bridge (SNB), Ivy Bridge (IVB) and Haswell (HSW) processors, there are documented PMU errata, respectively BJ122 [24], BV98 [25], HSD129 [26], which cause silent corruption of counts when Hyper-Threading is enabled.

Name	Code	Description
`MEM_UOPS_RETIRED.*`	0xd0	Memory μ-ops retired
`MEM_LOAD_UOPS_RETIRED.*`	0xd1	Load micro-ops retired
`MEM_LOAD_UOPS_LLC_HIT_RETIRED.*`	0xd2	L3 load hits retired
`MEM_LOAD_UOPS_LLC_MISS_RETIRED.*`	0xd3	L3 load misses retired

Figure 4.1: Corrupting events for SNB, IVB and HSW

We define *sibling threads* as hyper-threads sharing the same physical core. We also define *sibling counters* as the counters with the same index in the PMU of the sibling thread. Hereafter, we refer to the hyper-thread j as HT_j and to a PMU counter with index i as C_i. If certain memory events, listed in Table 4.1, are measured on C_i of one hyper-thread, they may corrupt any event measured on C_i of the sibling thread at the same time. In other words, the *crosstalk* corruption occurs between sibling counters.

Fig. 4.2 shows the possible combinations for corrupting (C) and non corrupting (NC) events on sibling threads HT_0 and HT_1 with 4 counters. The direction of the arrows indicates which counter is corrupting its sibling. For instance, C_1 of HT_0 is corrupting C_1 of HT_1.

The corruption causes over-counting on the impacted counter. The severity of the corruption cannot be predicted. It depends on the workload and the events measured on both hyper-threads. Event scheduling is not synchronized between

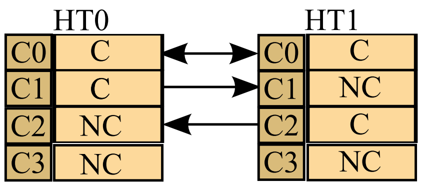


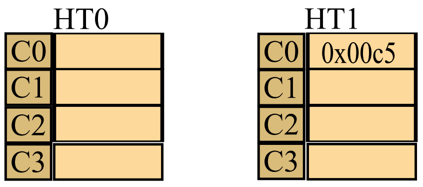
Figure 4.2: Possible counter corruptions between sibling threads

hyper-threads, hence, events can be programmed in and out of counters in any order and at any one time relative to the other hyper-thread making it even harder to predict the corruption error.

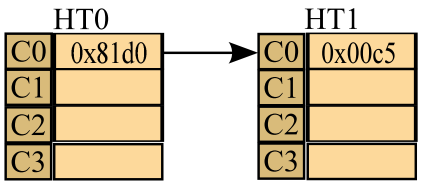
4.2 Corruption examples

The problem is very severe when a high rate memory event is leaking into a low rate event. To demonstrate this case, we run a simple memory intensive workload on an Intel Haswell client processor where logical CPU0 and logical CPU4 are sibling threads (threads HT_0 and HT_1 respectively). We use triad [10] workload, which computes $c[i] = a[i] + k * b[i]$, and we measure the corrupting MEM_LOAD_RETIRED.ALL_LOADS event (code 0x81d0) and the non-corrupting BRANCH_MISPREDICTION event (code 0x00c5). We expect the value of the branch misprediction event to be low, since it does not occur in tight loops. However, if the two events are measured on sibling counters of the sibling threads HT_0 and HT_1, then the corruption of the branch misprediction event can be orders of magnitude.

We use the perf tool for all of our experiments. It accepts raw PMU events using the `rXXXX` notation where `XXXX` is the hexadecimal event code.



(a) Add 0x00c5 on Thread 1



(b) Add 0x81d0 on Thread 0

Figure 4.3: Event assignment on CPU0 and CPU4

First, we measure the branch misprediction event only at the user level (`:u` modifier) on CPU4 (`-C` option), in system-wide mode (`-a` option) for 10 times (`-r` option), while also running the same triad test on CPU0 to generate the same load on each hyper-thread. We pin the triad program on CPU4 using the taskset tool:

```
$ taskset -c 0 triad &
$ perf stat -r 10  -a -C 4 -e r00c5:u taskset -c 4 triad
  644  r00c5:u
```

This generates the counter assignment shown in Fig. 4.3a and we get 644 as the total count for the branch misprediction event. Then, we add the measurement of the corrupting retired loads event on CPU0 (sibling thread):

```
$ perf  stat -a -C 0 -e r81d0:u taskset -c 0 triad &
$ perf stat -r10 -a -C 4 -e r00c5:u taskset -c 4 triad
  40,960,843 r00c5:u
```

We get the counter assignment shown in Fig. 4.3b and the measurement output is now 40,960,843 for the branch misprediction event instead of 644.

Comparing the results of the two tests, we see that the over-count is more than 60,000 times greater. This leads to very serious misinterpretation of the behavior of the workload, as triad may be assumed to be penalized by branch misprediction when in fact it is not.

To demonstrate that the problem is not specific to the branch mispredictions event, we run the example with another event for which we can figure out the count in advance such as ROB_MISC_EVENT.LBR_INSERTS (code 0x20cc) which counts the number of entries inserted in the Last Branch Record (LBR) buffer [27]. The event assignment is similar to the one in Fig. 4.3b, except for the event code in C_1. As the LBR is not used on the test system, the count for this event must be zero.

```
$ perf  stat -a -C 0 -e r81d0:u taskset -c 0 triad  &
$ perf stat -r 10 -a -C 4 -e r20cc:u taskset -c 4 triad
  41,284,632 r20cc:u
```

However, instead of zero the measurement output is 41,284,632. There is again a huge corruption due to the large number of load occurrences on the sibling thread. Note that the corrupted measurements of the non-corrupting LBR and branch misprediction events are about the same, which is reasonable given that the workloads are identical and the corrupting event measured on the sibling counter is the same.

There is also an error for the count of the corrupting event because it misses the counts that it leaked into the sibling counter. However, this error is small. To show that, we use again our second example but this time by also gathering the counts of the corrupting event on CPU0:

```
$ taskset -c 0 triad  & taskset -c 4 triad &
$ perf stat -a -C 0 -e r81d0:u sleep 5 &
$ perf stat -a -C 4 -e r20cc:u sleep 5
  2,827,304,988      r81d0:u
     71,654,800      r20cc:u $
```

The corrupted count of the non-corrupting event 0x20cc is what leaked from the count of the corrupting event 0x81d0. Thus, the total count for the event 0x81d0 would be the sum of these two counts. If we compare the ratio of the leaked count with the total count of the event 0x81d0, we get a ratio of 2.5%. In other words, missing counts of the corrupting event are negligible.

To prove that the corruption occurs only between sibling counters, we run two instances of the triad program pinned to CPU0 and CPU4 respectively. We invoke perf for a single measurement on the two hyper-threads. We use fewer events than there are counters, thus the counter assignment remains constant throughout the run and no rescheduling is needed. We use perf with the -a -A options to get a per logical CPU breakdown of the counts:

```
$ taskset -c 0 triad  & taskset -c 4 triad &
$ perf stat -a -C0,4 -A -e r81d0:u,r20cc:u sleep 5
  CPU0         2,823,288,122      r81d0:u
  CPU0                     0      r20cc:u

  CPU4         2,823,018,913      r81d0:u
  CPU4                     0      r20cc:u
```

On CPU4, there is no corruption on the LBR event: the count is zero as expected. The counter assignment for this example is shown in Fig. 4.4. In other words, C_0 of HT_0 does not corrupt C_1 of HT_1.

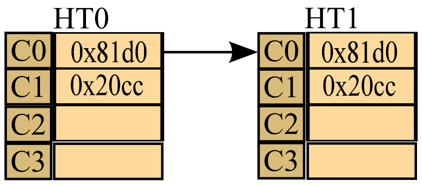


Figure 4.4: Simultaneous measurements on CPU0 and CPU4

It should be noted that in the configuration of Fig. 4.4, C_0 of HT_0 does corrupt C_0 of HT_1, as shown by the arrow, because corrupting events can corrupt each other across sibling threads. This is not a problem though, because the corruption is always small relative to the corrupting event total count as we have already demonstrated.

4.3 Possible solutions

The problem is severe and needs to be addressed. Several potential software workarounds have been discussed or implemented. However, these solutions avoid the problem by either preventing simultaneous measurements on sibling threads or by preventing the corrupting events from being measured. We present these workarounds below.

4.3.1 Disable Hyper-Threading

An obvious approach is to disable Hyper-Threading. This not only has a non-negligible performance impact for most workloads, but it is also impractical as Hyper-Threading can only be turned on/off on reboot, and not just when a corrupting event is measured.

4.3.2 User warnings

Users can be warned to measure only on one thread per physical core. This requires knowledge of the CPU topology and tools which can operate on subsets of processors. However, it is not very practical because it assumes a single measurement and single user machine; otherwise there is still a risk of corruption depending on what other users measure on the sibling threads. Furthermore, when measuring in per-thread mode, the problem still remains because program threads can migrate.

4.3.3 Monitor one thread per physical core

The kernel can ban half of the CPUs from monitoring. Only one hyper-thread per physical core can be used by the monitoring tools. Applications could still run on all logical CPUs but only half could be monitored. Again, not a very practical solution, especially if all threads of an application do not execute the same code.

4.3.4 Multiplex PMU measurements

The kernel can multiplex measurements between the two hyper-threads. Only one PMU per physical core can be active at any one time. This implementation in perf_events would be complex because it would require mutual exclusion logic between CPUs for event scheduling, something that is not there today. More importantly though, it would require turning off the hardware watchdog, which constantly uses a counter to sample on cycles event and trigger a non-maskable interrupt (NMI) to detect CPU deadlock. However, disabling the watchdog is not acceptable in many production environments where it is used for postmortem analysis of kernel deadlocks.

4.3.5 Ban corrupting events

The kernel can simply prevent the use of any corrupting events. This is, in fact, the current solution in the upstream Linux kernel running on Intel IvyBridge.

But the corrupting events which are shown in Table 4.1 and which are banned in the kernel as shown in the code above, are all very important memory events, needed in any serious performance analysis tools such as Gooda [10]. Thus, banning them completely is not a viable solution.

```
static struct event_constraint intel_ivb_event_constraints[] __read_mostly =
{
    ...

    /*
     * Errata BV98 -- MEM_*_RETIRED events can leak between counters of SMT
     * siblings; disable these events because they can corrupt unrelated
     * counters.
     */
    INTEL_EVENT_CONSTRAINT(0xd0, 0x0), /* MEM_UOPS_RETIRED.* */
    INTEL_EVENT_CONSTRAINT(0xd1, 0x0), /* MEM_LOAD_UOPS_RETIRED.* */
    INTEL_EVENT_CONSTRAINT(0xd2, 0x0), /* MEM_LOAD_UOPS_LLC_HIT_RETIRED.* */
    INTEL_EVENT_CONSTRAINT(0xd3, 0x0), /* MEM_LOAD_UOPS_LLC_MISS_RETIRED.* */
};
```

Figure 4.5: Intel IvyBridge corrupting event blacklisting

Chapter 5

Solving PMU Hardware Erratum: XSU Protocol

A protocol has been developed in perf_events subsystem to allow corrupting events to be used while avoiding any random corruption to the sibling thread counts. This protocol achieves fine-grained counter-level control between hyper-threads by enforcing mutual exclusion for sibling counters measuring the corrupting events and allowing sibling counters to measure simultaneously non-corrupting events.

Our solution leverages the perf_events event-oriented interface and the fact that event scheduling and multiplexing are entirely controlled by the kernel. Mutual exclusion between hyper-threads is achieved through coordinated event scheduling.

5.1 Dynamic event constraints

As described earlier, event scheduling is based on static event constraints implemented as bit masks. Our solution introduces *dynamic* event constraints. The static constraint, i.e., the hardware imposed constraint, is combined with a new constraint based on what is measured on the sibling thread at the same time to form the dynamic constraint. The perf_events scheduler operates on the dynamic constraints without any modification.

The key innovative idea in our solution is to leverage existing protocols maintaining cache coherence to generate the dynamic constraints for each event. The inspiration for this comes from the fact that there is an analogy between the counter corruption problem and cache line inconsistency issues in multi-processor systems. A counter pair, i.e., two sibling counters, corresponds to a cache line and the sibling threads accessing the counter pair correspond to the processors accessing the cache line.

The protocol we have developed is called XSU. It uses three states required for any hyper-thread in order to distinguish which PMU counters can perform measurements without yielding corrupting results. The three states are *exclusive* (X), *shared* (S) and *unused* (U). It is similar to the MSI cache coherence protocol [28], an invalidation-based protocol for write-back caches, but with fewer state transitions, as shown in Fig. 5.1.

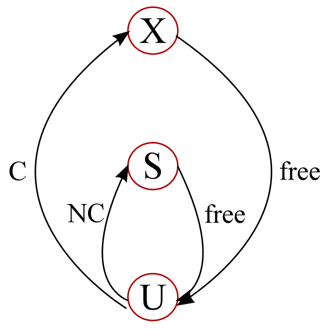


Figure 5.1: XSU state transition diagram

Initially, there is no event scheduled on the counter pair and it is marked unused (U state). This state corresponds to the Invalid state of MSI where no processor has a valid copy of the cache line. If a non-corrupting event is scheduled on one of the sibling counters, the state of the pair changes from unused to shared (S state). In the S state, another non-corrupting event can be measured simultaneously on the counter pair, i.e., sharing of the pair is allowed between the hyper-threads. This corresponds to the Shared state of MSI where more than one processor may have a valid copy of the cache line in their caches. If a corrupting event is scheduled on one of the sibling counters, the state of the pair changes from unused to exclusive (X state). This implies that no other event can be measured on the counter pair. To achieve measurement correctness, a corrupting event measurement requires exclusive use of a counter pair. The X state corresponds to the Modified state of MSI where only one processor has a valid copy of the cache line in its cache.

The transitions from the S and X states back to the U state happen when the events are scheduled out of the counters. During measurements, the pair is set to either S or X state and no other transitions are possible. To summarize, for each event type, the following is required:

- Corrupting: Allowed on counters in U state

- Non-Corrupting: Allowed on counters in U or S state

The kernel maintains the *XSU* state for each counter in a new data structure accessible from both hyper-threads as shown in Fig. 5.2.

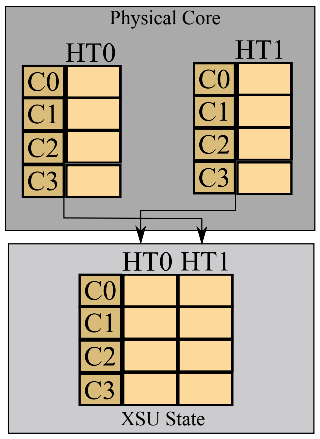


Figure 5.2: XSU shared state structure

5.2 Dynamic event scheduling

During scheduling, the XSU state of the requesting hyper-thread is read and combined with the static constraint of each event. The dynamic constraint mask of an event is simply the logical AND between the static constraint and the constraint mask built from the XSU counter state on the requesting hyper-thread. The bit mask constraint for each event type is constructed as shown in Fig 5.3.

	Corrupting	Non-corrupting
X	0	0
S	0	1
U	1	1

Figure 5.3: XSU constraint bitmask

Based on its dynamic constraint, the event is assigned to a counter on the requesting hyper-thread. Then, the XSU state of the sibling (non-requesting) hyper-thread is modified to reflect what it can measure after scheduling is complete.

In order to demonstrate how the XSU protocol operates, let us assume that the kernel needs to schedule a list of one corrupting (C) and two non-corrupting (NC) events on HT_0 and a list of one non-corrupting and two corrupting events on HT_1, where HT_0 and HT_1 are sibling threads.

For simplicity, let us also assume that all events are statically unconstrained. With 4 counters, the static constraint is therefore 0b1111. The dynamic constraint of each event, as the logical AND of its static constraint and the XSU state constraint is therefore equal to the XSU state constraint. Note that at the formulation of the constraints, C_0 is represented to the least significant bit and C_3 to the most significant bit.

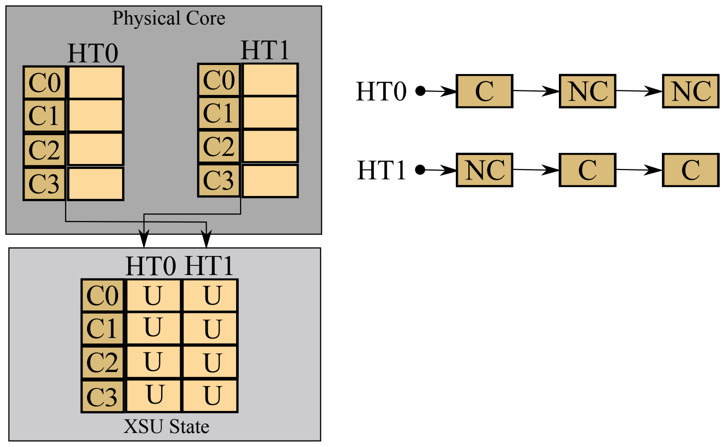


Figure 5.4: Initial State

As shown in Fig. 5.4, initially the PMU counters of both sibling threads are empty and the XSU state for each of them is marked as U.

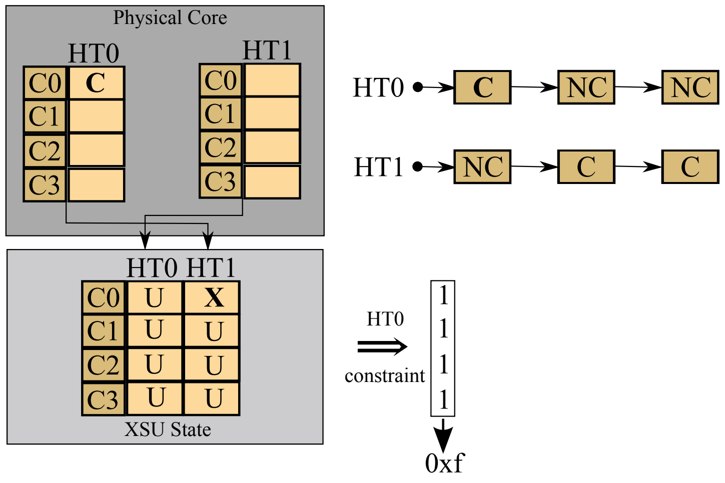


Figure 5.5: Scheduling first event (C) of HT_0 list

The kernel starts with the first event (C) of HT_0. The dynamic constraint is 0b1111 as all the PMU counters of HT_0 are marked as U. Thus, this event can run on any counter of HT_0. As shown in Fig. 5.5, the scheduler chooses the first available counter and the event is scheduled on C_0. The XSU state of C_0 of HT_1 is updated to X. This implies that C_0 can no longer be used for any event measurement by HT_1 because of the corrupting event measured by HT_0 on its sibling counter.

In Fig. 5.6, the kernel proceeds with the first event (NC) of HT_1. The dynamic constraint is now 0b1110 because of C_0 marked as X. Thus, this event can run

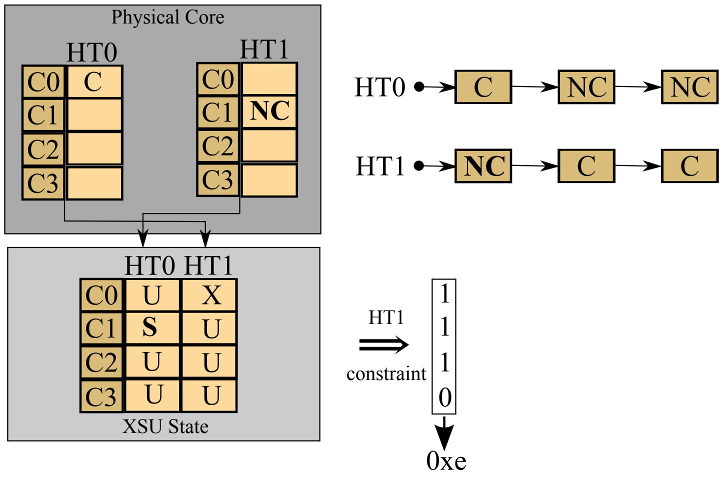


Figure 5.6: Scheduling first event (NC) of HT_1 list

on one of C_1, C_2 or C_3 of HT_1. The event is scheduled on the first available which is C_1. The XSU state of C_1 of HT_0 is updated to S and HT_0 can only use this counter for measuring a non-corrupting event because measuring a corrupting would impact the measurements of the NC event on the sibling C_1 of HT_1. This implies that C_0 can no longer be used for any event measurement by HT_1 because of the corrupting event measured by HT_0 on its sibling counter.

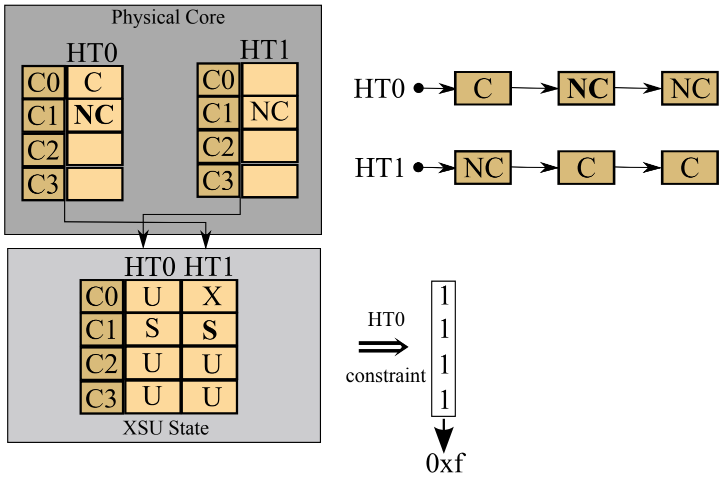


Figure 5.7: Scheduling second event (NC) of HT_0 list

Next comes the second event (NC) of HT_0 as shown in Fig. 5.7. The dynamic constraint is 0b1111 since, static constraint permitting, a non-corrupting event can be scheduled on any counter that is in shared or unused state. The event is scheduled on C_1, the first available counter of HT_0. The XSU state of C_1 of HT_1

is updated it to S in order to reserve the counter only for non-corrupting events.

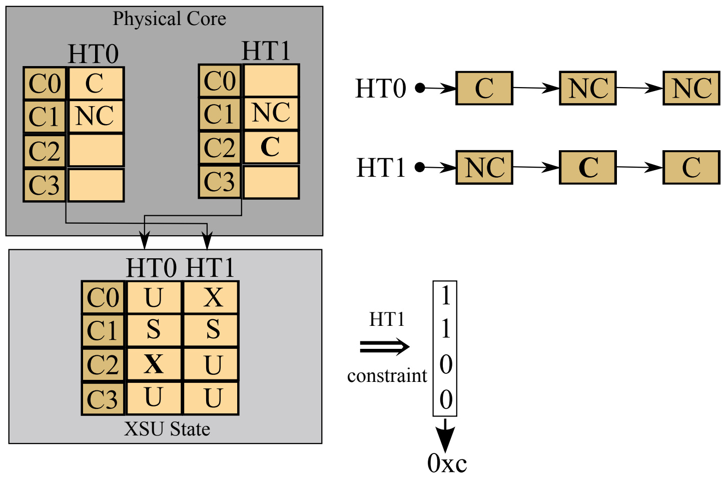


Figure 5.8: Scheduling second event (C) of HT_1 list

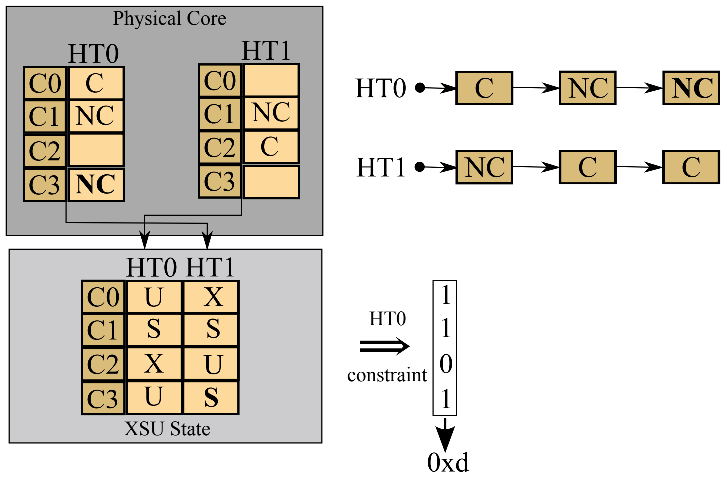


Figure 5.9: Scheduling third event (NC) of HT_0 list

In Fig. 5.7 and Fig. 5.8 the kernel proceeds with the scheduling of the third event (NC) of HT_0 and the second event (C) of HT_1 based on the XSU protocol principles. Now, let us focus on the last event (C) of HT_1.

A corrupting event can be scheduled only on counters that are in unused state, static constraint permitting. Therefore, as shown in Fig. 5.10 the event can only be scheduled on C_2 and the dynamic constraint is 0b0100. However, C_2 of HT_1 is occupied by another event. Thus, the last event in the list of HT_1 cannot be scheduled. As described, in chapter 3 this will induce multiplexing. When an event

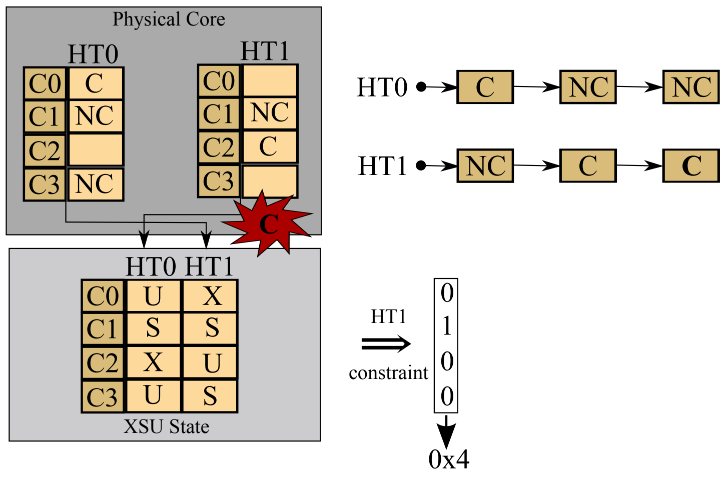


Figure 5.10: Scheduling third event (C) of HT_0 list

is eventually scheduled out, the counter is freed and its XSU state is changed back to the U state in the sibling thread.

5.3 Integrating XSU protocol in perf_events subsystem

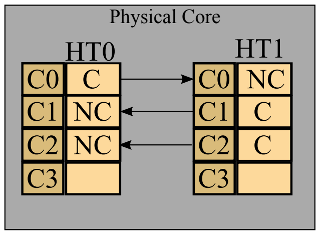
5.3.1 Advantages

The integration of the XSU protocol in perf_events guarantees measurement correctness. At the same time, it enables measurements of all the performance events, including the corrupting memory events which are needed for any serious performance analysis. It makes it possible to reliably use Intel processors' PMU counters and features for advanced performance analysis by tools such as Perf and Gooda. The XSU protocol has several key advantages compared to the solutions discussed in section 4.3 of the previous chapter.

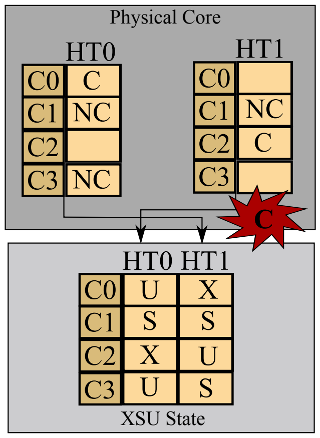
- Hyper-Threading remains enabled.
- There are no changes at the user level, so existing collection scripts and tools do not need to be modified.
- No event is blacklisted.
- All logical CPUs can be monitored simultaneously.

5.3.2 Tradeoffs

When the XSU protocol is used, there is necessarily more pressure on event scheduling. Common events which were not constrained before may become constrained dynamically, depending on what is measured on the sibling thread. Scheduling becomes more difficult as more corrupting events appear in the event lists of the sibling threads.



(a) No multiplexing but corrupted results



(b) Correct results but multiplexing

Figure 5.11: XSU protocol effects

Back to the example we saw above, as the Fig. 5.11 demonstrates, without the XSU protocol all the events can be scheduled in one scheduling pass but corrupted results will be yielded for all the NC events. Whereas with the XSU protocol, measurement correctness is now guaranteed but multiplexing is needed.

Under certain conditions, it is quite possible for the dynamic constraint mask to come out as zero, meaning that the event cannot be currently scheduled. This also induces multiplexing which will give a chance for the event to be scheduled later.

If no corrupting event is programmed, our solution does not modify existing event scheduling constraints and it only incurs the extra book-keeping cost of the XSU state structure.

The key implementation challenge is to tune the XSU protocol with the incremental perf_events scheduler. This can happen assuming the dynamic constraints as constant for a given events list only for as long the resources are atomically acquired by the specific thread.

5.3.3 Implementation

We have successfully implemented our XSU protocol in Linux kernel 3.15. We have modified 5 files and about 600 lines of code. The code has been published on the Linux kernel mailing list (LKML) and at the time of this writing is under review by subsystem maintainers.

The implementation was relatively straightforward. We added the shared XSU state structure to each pair of threads and protected it with a spinlock.

The corrupting events are added to the constraint table for each processor with the erratum using the `INTEL_EXCLEVT_CONSTRAINT()` macro as shown in Fig. 5.12 for the Haswell processor.

```
static struct event_constraint intel_hsw_event_constraints[] = {
    ...

    INTEL_EXCLEVT_CONSTRAINT(0xd0, 0xf), /* MEM_UOPS_RETIRED.* */
    INTEL_EXCLEVT_CONSTRAINT(0xd1, 0xf), /* MEM_LOAD_UOPS_RETIRED.* */
    INTEL_EXCLEVT_CONSTRAINT(0xd2, 0xf), /* MEM_LOAD_UOPS_LLC_HIT_RETIRED.* */
    INTEL_EXCLEVT_CONSTRAINT(0xd3, 0xf), /* MEM_LOAD_UOPS_LLC_MISS_RETIRED.* */

    ...
};
```

Figure 5.12: Intel Haswell corrupting event constraints encoding

Each time an event is used, the table is looked up and if the matching event is found, then the constraint and some flags are extracted. For the corrupting events, the flags indicates that the event requires exclusive counter access via the XSU protocol. The constraint on the corrupting events is 0xf, i.e., any generic counter because those events do not have static hardware constraints.

When running on a processor with the erratum, all events must go through the XSU protocol to compute their dynamic constraint. If no corrupting event is present, then the regular constraint on the event is not modified. The XSU protocol code is specific to Intel PMU and there lives in the perf_event_intel.c file exclusively. No changes to the generic X86 perf_events code is required.

In order to ensure correctness of the dynamic constraints, the XSU shared state is locked during scheduling by either hyper-thread. Scheduling must appear as a atomic transaction to ensure that if scheduling succeeds the new XSU state can be committed safely, i.e., without the risk of the sibling having run and modified the state in the meantime. This locking is implemented through a new set of PMU specific optional callbacks. They are defined only for Intel X86 PMU models with the erratum. The rest of the scheduling algorithm is unmodified.

The dynamic constraints are built on the fly and require memory allocation per event. The new code ensures that memory is freed appropriately by flagging the events with dynamic constraints.

The code also provide a way to disable the workaround for debugging purposes via a sysfs file entry for the core PMU: /sys/devices/cpu/ht_bug_workaround. The workaround is enabled by default. To disable the workaround, a system administrator must do:

```
# echo 0 >/sys/devices/cpu/ht_bug_workaround
```

To re-enable the workaround:

```
# echo 1 >/sys/devices/cpu/ht_bug_workaround
```

We will evaluate the effect of the XSU protocol in perf_events measurement correctness with specific examples in chapter 7.

Chapter 6

Optimizing Event Scheduling

6.1 Drawbacks of existing scheduling algorithm

As described in chapter 3, the current Intel x86 algorithm uses a greedy, first match approach to assign events to counters. The first available counter which satisfies an event's constraint is selected for the event to be scheduled on. Once an event is assigned to a counter, it cannot be reassigned even though its constraint could permit its assignment on another counter.

The scheduling algorithm stops adding events at the first error, which occurs at most after N events, if there are N counters. However, in reality, the scheduler stops much earlier. It stops at the first error due to event constraints which cannot be satisfied simultaneously from the remaining set of free counters.

The major issue of the algorithm described is that the event scheduler may not choose wisely the counter to schedule the event on. In order to maximize the possibilities for the subsequent events to be scheduled, it needs to have knowledge of their constraints and make a counter selection based on that and not by choosing the first available counter. Furthermore, with N counters, the scheduling algorithm only uses the first N events of the linked list at each iteration, potentially leaving aside events which could use the counters left over by event conflicts. The confluence of these two factor leads to a suboptimal allocation of events to counters and results in under-utilization of the available resources or even failure to resolve scheduling problems of certain combinations of events and constraints.

As we have seen, in situations where not all events can be scheduled at once, the kernel multiplexes them, i.e., time-sharing of the counters is required. The fewer events scheduled by the scheduling algorithm, the more multiplexing is needed. However, with multiplexing, the events are not measured at all times because of the time-sharing. The kernel keeps track of the time the event was enabled versus the time it is actually ran on the PMU hardware. The information is passed back to the performance monitoring tools which then scale the event count based on the timing ratio. The computation is shown in equation 6.1 below.

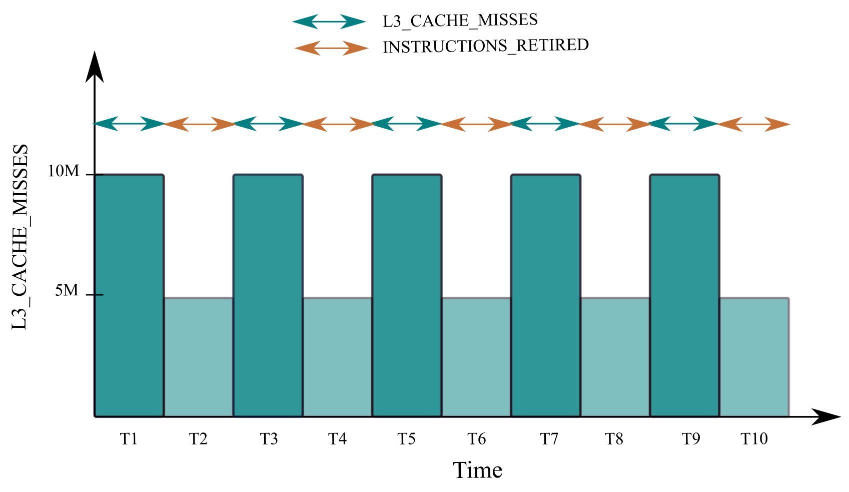
$$count = \frac{count_{raw} * time_{enabled}}{time_{running}} \tag{6.1}$$

In that case, the final count is an approximation of what it would have actually been, had the event been measured throughout the entire time it was indicated as enabled. This calculation works very well if the workload has a constant behavior, as it assumes the events always occur at the same rate.

However, if the workload has rapidly changing phases, as shown in Fig. 6.1, this scaling calculation could yield inaccurate results. Let us assume that there are two events measured, L3_CACHE_MISS and INSTRUCTIONS_RETIRED. Let us also assume that both events compete for the same counter. The measurements run for 10 equal time intervals of duration T, $T_1 = ...T_{10} = T$. The blue histogram corresponds to the counts of the cache misses events each on of these 10 time intervals. The blue slots correspond to the time intervals where the cache misses event is running on the counter while the beige slots correspond to the ones where the instructions event occupies the counter. The cache misses event is enabled for all the 10 time intervals, thus $time_{enabled} = 10T$. In reality, it runs on the hardware only for the slots T_1, T_3, T_5, T_7 and T_9, thus $time_{running} = 5T$. The total count that the kernel passes to the performance monitoring tools corresponds to the 5 slots the event was measured and it is $count_{raw} = 50$M. Based on equation 6.1 the scaled count we obtain for the cache misses event is 100M. However, if we add up the measurements for each interval of the histogram the actual count of cache misses is 75M. Thus, scaling of the counts due to multiplexing is not always reliable and can yield significant inaccuracies.

An obvious way to mitigate the error is to increase the rate of multiplexing, but that also increases the overhead of monitoring which is not desirable, as the behavior of the workload would be impacted. Another way is to minimize multiplexing and thus make the $time_{running}$ approximate the $time_{enabled}$, is by maximizing the use of the PMU counters. This requires that the scheduling algorithm stops as late as possible and maximizes the number of events scheduled at each run.

The existing Intel x86 scheduling does not always maximize counter usage when many events are constrained. The greedy algorithm works better when most events are not constrained. However, as discussed in section 5, the XSU protocol guarantees measurement correctness at the cost of more constrained scheduling. Although, the current event scheduling algorithm performs well when most events are unconstrained, the deterioration of the quality of schedules in a more constrained environment is unavoidable and raises the need of a more effective scheduling algorithm to solve the event/counter assignment.



a

Figure 6.1: Multiplexing induced inaccuracies

6.2 Event scheduling as graph matching

The perf_events scheduling can be modeled as a matching problem in an unweighted bipartite graph G. The events and the counters are the two disjoint sets of the bipartite graph (let us call them X and Y respectively) and they form the set V of vertices in the graph. The constraints allowing each event to be measured only on certain PMU counters, form the set E of edges in the graph. To schedule the events optimally on the hardware, we need to find the maximum bipartite matching in $G(V, E)$. This model is equivalent to adding a super source s with edges to all vertices in the events set X and a super sink t with edges from all vertices in the counters set Y, and finding a maximal flow from s to t. All edges with flow from X to Y then constitute a maximum matching.

6.3 Algorithms for graph matching

We now describe the theoretical background of graph matching problem algorithms as presented in [29].

Definition 6.3.1. *Let $G = (V, E)$ be a graph. $M \subseteq E$ is called **matching** of G if $\forall\ v \in V$ we have $|\ \{e \in M : v \text{ is incident on } e \in E\}\ | \leq 1$.*

Definition 6.3.2. *A matching M of G is called **maximal** if $\forall\ e \in E \setminus M$ the set of edges given by $M \cup \{e\}$ is not a matching of G.*

Definition 6.3.3. *The **size** of a matching M of G is the number of the edges it contains and is denoted by $|M|$.*

Definition 6.3.4. *A matching M of G is called **maximum** if $\forall$ matching M' of G we have $|M| \geq |M'|$.*

Definition 6.3.5. *Let M be matching of G. A vertex $v \in V$ is called **M-saturated** if M contains an edge incident on v. Otherwise v it is called **M-unsaturated** or **M-free**.*

6.3.1 Augmenting paths

Definition 6.3.6. *Let M be matching of G. A path P in G is called **M-alternating** if the edges of P are alternately in and out of M.*

Definition 6.3.7. *Let M be matching of G. A path P in G is called **M-augmenting** if it is a maximal, M-alternating path with unsaturated start and end vertices. Clearly, an M-augmenting path has odd number of edges.*

Lemma 6.3.8. *Let G be a graph whose maximum degree is at most 2. Then every component of G is either an isolated point, a path or a cycle.*

Proof. Consider any non-isolated vertex v of G. Its, at most two, neighbors further have degree at most 2 and so on. So the component of G containing v is either a path or a cycle. This holds true for all non-isolated vertices of G. QED. □

Lemma 6.3.9. *(Berge 1957) A matching M is maximum if and only if G has no M-augmenting path.*

Proof. Suppose there exists an M-augmenting path P. Consider the symmetric difference $M \oplus P$ which represent edges that are present in exactly one of M or P. Since P is an M-augmenting path, $M \oplus P$ is also a matching of G and $|M \oplus P| = |M| + 1$. So M is not maximum. Suppose M is not maximum. Let M' be a maximum matching and so we have $|M'| > |M|$. Consider $M \oplus M'$. Each vertex has degree at most 2 in $M \oplus M'$ since each of M and M' can contribute at most 1 each to the degree of each vertex in $M \oplus M'$. By Lemma 6.3.8, $M \oplus M'$ consists of cycles, paths and isolated vertices. But the edges of $M \oplus M'$ alternate in M and M' exclusively. Hence each cycle must be even. So M' exceed M in size only from the paths. So, there exists at least one path in $M \oplus M'$ which has more edges from M' than from M. But such a path is M-augmenting. □

Corollary 1. *(Hopcroft-Karp) Let M^* be a matching of G. Then for any matching M of G such that $|M^*| \geq |M|$, we have $|M^*| - |M|$ vertex-disjoint M-augmenting paths. The non-M edges on these paths all belong to M^*.*

Proof. From the proof of 6.3.9 every cycle of $M \oplus M^*$ is even and every path of $M \oplus M^*$ which is not M-augmenting must have equal number of edges from M and M^* as M^* is maximum. Also note that each M-augmenting path has exactly one edge more from M^* than from M. So we need $|M^*| - |M|$ such paths. These are all vertex-disjoint, since in Definition 6.3.7 we defined augmenting paths as maximal paths starting and ending at unsaturated points. □

Corollary 2. *Let $|M^*|$ be a maximum matching and M be any matching. If M is not maximum, then the shortest M-augmenting path has length $\leq \frac{|V|}{|M^*|-|M|} - 1$*

Proof. From Corollary 2 we know that there are $|M^*| - |M|$ vertex-disjoint (and hence edge-disjoint) M-augmenting paths. By Pigeonhole Principle, one of the paths must have at most $\frac{|V|}{|M^*|-|M|}$ vertices and thus has length at most $\frac{|V|}{|M^*|-|M|} - 1$. □

For finding the maximum matching of a graph using augmenting paths, consider the following algorithm which follows from Lemma 6.3.9.

```
1. $M = \emptyset$
2. **while** (there is an $M$-augmenting path $P$) **do**
       $M \leftarrow M \oplus P$
3. **return** $M$
```

The challenge now is to detect existence of and find augmenting paths efficiently. We consider the case when G is bipartite.

6.3.2 An $O(n^3)$ algorithm for finding maximum matching in bipartite graphs

Let $G = (X \cup Y, E)$ be a bipartite graph where X and Y are its disjoint sets and let M be a matching of G. We want to find a maximum matching of G. We denote by X_0 , Y_0 the sets of M-unsaturated vertices in X, Y respectively. We consider a new directed graph H on the vertex set $X \cup Y$ and edge set E. Edges which are in M are directed $X \to Y$ and edges not in M are directed $Y \to X$.

Lemma 6.3.10. *G has a M-augmenting path if and only if H has a path from Y_0 to X_0.*

Proof. Suppose G has an M-augmenting path say from $u \in X_0$ to $v \in Y_0$. The same path directed from v to u is clearly a path in H from Y_0 to X_0. Suppose H has a path from $y \in Y_0$ to $x \in X_0$. The underlying undirected path from x to y is clearly an M-augmenting path. □

So we do a depth-first-search (DFS) from Y_0 and stop as soon as we reach some vertex in X_0, thus giving us an M-augmenting path P. M is augmented along P, the new matching is $M \oplus P$. The process is then repeated. If a vertex of X_0 cannot be reached, then G has no M-augmenting path i.e. M is maximum. The time complexity of the algorithm is analyzed as follows.

1. Without loss of generality, assume $|Y| \leq |X|$. Thus $|Y_0| \leq |Y| \leq \frac{|V|}{2}$. Also at each stage of the algorithm, augmenting saturates a previously unsaturated vertex from Y without impacting vertices which are already saturated. So we need at most $|Y_0| \leq \frac{|V|}{2}$ stages.

2. At each stage several DFS are needed, starting from each vertex in Y_0. The maximum number of DFS needed is $|Y_0|$, as in the worst-case, only the DFS starting from the last vertex of Y_0 may lead to a path in X_0. A single DFS can be done in $O(|V| + |E|)$ time.

3. Once an augmenting path is found, the matching is augmented in $O(|E|)$ time.

Thus, the algorithm takes at most $\frac{|V|}{2}\Big[O(|E|) + |Y_0| * O(|V| + |E|)\Big]$. However, the $|Y_0|$ factor can be eliminated. If a super-vertex ψ is added and connected with edges to to every point in Y_0, DFS is needed to be applied only once, for vertex ψ. Thus, the time complexity becomes $O\Big(\frac{|V|}{2}\Big[|E| + (|V| + |E|)\Big]\Big) = O\Big(|V|^2 + |V||E|\Big)$. Since a bipartite graph on $|V|$ vertices can contain at most $(\frac{|V|^2}{4})$ edges, the time complexity of the algorithm is $O(|V|^3)$ or $O(n^3)$.

6.3.3 Hopcroft-Karp algorithm for finding a maximum matching in bipartite graphs in $O(n^{2.5})$ time

The preceding algorithm, looked for a single augmenting path at a time and augmented it. The maximal family of vertex-disjoint shortest-length augmenting paths could be found instead and be augmented all together in a single stage. This would bring the time complexity down to $O(n^{2.5})$. Consider the following algorithm.

```
1. $M = \emptyset$

2. **while** (there is an $M$-augmenting path) **do**
      find a maximal family $F$ of vertex-disjoint shortest $M$-augmenting paths;
      set $M \leftarrow M \oplus F$;

3. **return** $M$
```

The correctness of the algorithm follows from Lemma 6.3.9. It can be shown that using a maximal family F of shortest augmenting paths instead of a single augmenting path significantly reduces the number of stages (Lemma 6.3.14), and also that the time per stage induced by finding such families does not increase (Lemma 6.3.15). The proof of the above is based on the following lemmas.

Lemma 6.3.11. *Let M be a matching of G and let P be an M-augmenting path of shortest length. Let P' be an $(M \oplus P)$-augmenting path. Then $|P'| \geq |P| + |P \cap P'|$, where $|P|$ is the number of edges in P.*

Proof. Consider $N = (M \oplus P) \oplus P'$. Then N is clearly a matching and $|N| = |M| + 2$. Thus by Corollary 1, there are 2 vertex-disjoint M-augmenting paths, say P_1 and P_2, with the non-M edges in N. That is, $P_1 \cup P_2 \subseteq M \oplus N$. Note that $M \oplus N = P \oplus P'$ and thus we have $|P \oplus P'| \geq |P_1| + |P_2|$. But P_1, P_2 are both M-augmenting paths and P is a shortest M-augmenting path. Therefore $|P \oplus P'| \geq 2|P|$. However $|P \oplus P'| = |P| + |P'| - |P \cap P'|$ and so the desired inequality follows. $\square$

Lemma 6.3.12. *Let $M_0 = \emptyset$ and consider the sequence M_0, M_1, ..., M_i, ... where $\forall i$, P_i is a shortest M_i-augmenting path, and $M_{i+1} = M_i \oplus P_i$. Then, for $i < j$, $|P_i| \leq |P_j|$. Further, $|P_i| = |P_j|$ implies that P_i and P_j are vertex-disjoint.*

Proof. It follows from Lemma 6.3.11 that for $i < j$, $|P_i| \leq |P_j|$. Suppose now that for some $i < j$, P_i and P_j are not vertex-disjoint, and assume to the contrary that $|P_i| = |P_j|$. This implies that $|P_i| = |P_{i+1}| = ... = |P_{j-1}| = |P_j|$. Then there exist some k, l such that $i \leq k < l \leq j$ and P_k and P_l are not vertex-disjoint and further for all m between l and k we have P_m is vertex-disjoint from both P_k and P_l. Therefore P_l is an M_k-augmenting path and so by Lemma 6.3.11 we have $|P_l| \geq |P_k| + |P_l \cap P_k|$. However we are given that $|P_l| = |P_k|$ which implies that $|P_l \cap P_k| = 0$, i.e., P_l and P_k have no edges in common. However since P_l and P_k are not vertex-disjoint, they have a common vertex say x and then they must have in common the edge from $M_k \oplus P_k$ which is incident on x leading to a contradiction. $\square$

Lemma 6.3.13. *Let F be an inclusion-maximal family of vertex-disjoint shortest M-augmenting paths, all of length l_1. Let l_2 be the length of a shortest $(M \oplus F)$-augmenting path. Then $l_2 \geq l_1 + 2$.*

Proof. Let $F = \{P_1, P_2, ..., P_r\}$. Let P be a shortest $(M \oplus F)$-augmenting path. Note that $M \oplus F = (...(M \oplus P_1) \oplus P_2)...) \oplus P_r$. Suppose P is disjoint from each element of F. Then P is also an M-augmenting path, but by maximality of F, it is not a shortest augmenting path. So $l_2 > l_1$. Next, suppose that P has a vertex in common with at least one path in F. By Lemma 6.3.12 we have $l_2 > l_1$. Finally note that l_1, l_2 are both lengths of augmenting paths and they must be odd; hence $l_2 > l_1 \implies l_2 \geq l_1 + 2$. $\square$

We consider again the bipartite graph $G = (X \cup Y, E)$ where X and Y are its disjoint sets and the directed graph H with vertex set $X \cup Y$ and edge set E.

Lemma 6.3.14. *The algorithm described at the start of this section makes at most $2\sqrt{|V|}$ iterations.*

Proof. Let M^* be a maximum matching and let M be the matching after $\sqrt{|V|}$ iterations. By Lemma 6.3.13, the length of the shortest M-augmenting path is at least $(2\sqrt{|V|} - 1) \geq \sqrt{|V|}$. By Corollary 2 we have $\sqrt{|V|} \leq$ (length of shortest M-augmenting path) $\leq \frac{|V|}{|M^*|-|M|}$, and so $|M^*| - |M| \leq \sqrt{|V|}$. From this point onwards, even if we augment just one path in each iteration, we need at most $\sqrt{|V|}$ more iterations, as each augmentation increases size of matching by 1. Thus overall we need no more than $2\sqrt{|V|}$ iterations. $\square$

Lemma 6.3.15. *Each iteration of the algorithm can be implemented in $O(|E|)$ time.*

Proof. First we will use breadth-first-search BFS to find the length k of a shortest path from Y_0 to X_0. Simultaneously, we produce the sequence of disjoint layers $Y_0 = L_0, L_1, ..., L_k \subseteq X_0$ where

- $\forall i : 0 \leq i < k$, L_i is the set of vertices at distance i from Y_0
- L_k is the subset of X_0 at distance k from Y_0 which we look for

To avoid multiple BFSs from each vertex in Y_0, a super-vertex ψ is added with edges connecting it to all vertices of Y_0. The distance of ψ from X_0 can be found with a BFS starting from ψ. Subtracting this by one gives the length of the shortest path from Y_0 to X_0. This requires $O(|E|)$ time. Now consider a modified DFS which starts at a vertex $v \in Y_0$, stops as soon as it reaches a vertex say w in L_k and outputs this $v \to w$ path. Add this M-augmenting path to F and delete all vertices visited in the modified DFS. This is crucial; not just the augmenting path is deleted but also all the other vertices visited in the modified DFS. Let x be a vertex seen at some L_j in the DFS started from $v \in Y_0$. If x does not lead to an M-augmenting path of length k starting at v, then x cannot be on any M-augmenting path of length k: any such path has to begin at some vertex in Y_0 and it has to use i edges to reach x. If the procedure is repeated starting at another vertex in Y_0 until all vertices of Y_0 are explored, a maximal family of vertex-disjoint shortest-length augmenting paths is found. Let m_i be the number of edges visited in the i^{th} DFS which takes $O(m_i)$ time. Noting that $|E| \geq \sum_i m_i$, the time taken is $O(|E|)$. □

Theorem 6.3.16. *The algorithm runs in $O(|V|^{2.5})$ time.*

Proof. From Lemma 6.3.15 each phase can be implemented in $O(|E|)$ time. Also from Lemma 6.3.14 there are at most $2\sqrt{n}$ phases. Thus, the time complexity of the algorithm is $O(\sqrt{|V|}) * O(|E|) = O(|V|^{2.5})$. □

6.4 Integrating Hopcroft-Karp in perf_events scheduler

The Hopcroft-Karp maximum cardinality matching graph algorithm can replace the greedy, first match algorithm which is responsible for assigning the events on the PMU counters in the perf_events subsystem.

The pseudocode of the Hopcroft-Karp algorithm is demonstrated in Fig 6.2 that follows.

```
function Hopcroft-Karp() {
 for each (x in X)
  pair_Y[x] = free;
 for each (y in Y)
  pair_X[y] = free;

 matching = 0;

 while (BFS() == true)
  for each (x in X)
   if (pair_Y[x] = free)
    if (DFS(x) == true)
     matching = matching+1;

 return matching;
}
```

```
function BFS() {
 for each (x in X)
  if (pair_Y[x] = free) {
   distance[x] = 0;
   enqueue(Q, x);
  } else {
   distance[x] = ∞;
  }

 distance[free] = ∞;

 while (!empty(Q)) {
  x = dequeue(Q);

  if(distance[x] < distance[free])
   for each (y in Y)
    if (adjacent[x][y])
     if (distance[pair_X[y]] == ∞) {
      distance[pair_X[y]] = distance[x]+1;
      enqueue(Q, pair_X[y])
     }
 }

 return distance[free] != ∞;
}
```

```
function DFS(x in X) {
 if (x == free)
  return true;

 for each (y in Y)
  if (adjacent[x][y])
   if (distance[pair_X[y]] == distance[x]+1)
    if (DFS(pair_X[y])) {
     pair_Y[x] = y
     pair_X[y] = x
     return true;
    }

 distance[x] = ∞;
 return false;
}
```

Figure 6.2: Hopcroft-Karp Pseudocode

6.4.1 Tradeoffs

As we have shown in section 3.2, the complexity of the existing scheduling algorithm is $O(n^2)$ whereas the complexity of Hopcroft-Karp is $O(n^{2.5})$. However, n in our case is bound by the number of PMU counters.

In section 3.1 we have described that if the PMU has N counters, the generic layer passes at most N events down to the architecture specific layer of the scheduler. This guarantees that if an arbitrary long list of events is provided, the system will not slow down proportionally. This means that any scheduling instance has at most N events to be scheduled on the N counters. In the complexity analysis of the previous section n correspond to the number of vertices in the bipartite graph of the problem, i.e. the number of counters plus the number of events. All major PMUs nowadays have less than 16 counters. That number is very likely to remain low because adding generic counters in hardware is a very expensive proposition. Hence, in such small scheduling instances, the overhead of our higher complexity scheduling algorithm is negligible and is at scale of nanoseconds.

Our proposal guarantees optimal scheduling no matter how complicated the constraint might be, without making the event scheduling more time consuming. This is very critical, especially now with the integration of the XSU protocol in perf_events. Hopcroft-Karp can compensate for the more constrained scheduling and limit multiplexing. The advantage of our proposal versus the existing scheduling algorithm is demonstrated in the Fig. 6.3 below.

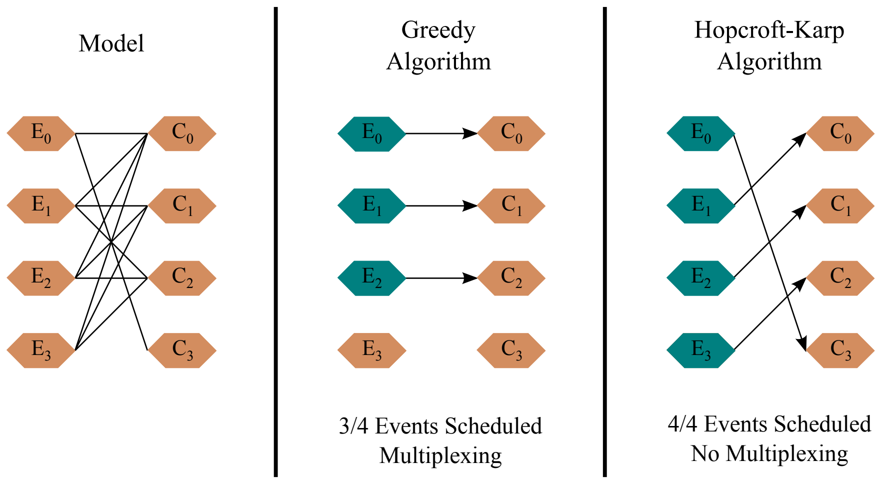


Figure 6.3: Hopcroft-Karp versus Greedy scheduling algorithm

6.4.2 Implementation

This high-level graph algorithm has been successfully implemented in the Linux kernel meeting all the requirements associated with kernel development.

Compact C Code : The Linux kernel is written in pure C and assembly. It operates under certain constraints regarding memory allocations. Data structures are usually fairly simple.

Minimal Execution Time : The event scheduling algorithm is invoked frequently. By default, at every timer tick (1ms on x86) when multiplexing is necessary.

It is also invoked on tasks context switches for per-thread events. The context switch code path is very latency sensitive, this is why the scheduling needs to be fast.

Minimal Memory Footprint : The Linux kernel tries to minimize its memory usage to maximize free memory for applications. In particular, the stack size very limited.

No Recursion : On kernel entry, the stack of each thread is switched from the expandable user stack to a per-thread fixed-size kernel stack. That stack is very limited in size usually two pages (8KB on x86). It cannot grow automatically like the user level stack. That prohibits recursive functions because stack consumption may not easily be predictable.

The code has been designed appropriately to solve the small scheduling instances of perf_events extremely fast while using simple enough data structures. The recursive modified DFS we demonstrated in section 6.3.3, has been redesigned to be iterative. For eliminating recursion in DFS, a stack is used in order to keep track of the vertices in Y reached by the X vertices produced by the BFS layer. The pseudocode of the iterative DFS is demonstrated in Fig. 6.4 below.

The algorithm is implemented in the x86 specific layer, and is shared by Intel and AMD processors. The actual file modified is arch/x86/kernel/cpu/perf_event.c and about 300 lines of code were added. The collection of event constraints and short path are not modified at all by the new code. Only the perf_assign_events() function is replaced. The bitmasks for the event constraints are decoded to build the graph on entry and the assignment is translated from the graph back into an array of integers on exit as expected by the calling function.

We will evaluate the the effect of the event scheduling optimization with specific examples in the chapter that follows.

```
function DFS(x inX) {
  if (x == free)
    return true;

  push(stack, (x, $y_0$));

  while (!empty(stack)) {
    x = top(stack).X;

    for each ($y_i$ in Y) {
      top(stack).Y = $y_i$

      if (adjacent[x][$y_i$])
        if (distance[x_pair[$y_i$] = distance[x] + 1) {
          if (x_pair[$y_i$] == free) {
            /*
             * A free vertex of X is reached and an augmenting path was found.
             * Pop the entire stack adjusting the pairs and return true.
             */
            while (!empty(stack)) {
              /* Adjust pairs */
              x_pair[top(stack).Y] = top(stack).X;
              y_pair[top(stack).X] = top(stack).Y;
              /* Pop stack */
              pop(stack);
            }
            return true;
          } else {
            /*
             * The new 'x' node is pushed in the stack for further check.
             */
            push(stack, (x_pair[$y_i$], $y_0$);
            }

          break;
        }

        /*
         * The 'x' node's neighbors in Y set were fully scanned and
         * nothing interesting was found: pop and continue with the
         * other candidates in X.
         */
        distance[x] = $\infty$;
        pop(stack);
      }
    }
```

Figure 6.4: Iterative DFS for kernel integration

Chapter 7

Evaluation

In this chapter, we evaluate the results of our solutions for both the XSU protocol and the Hopcroft-Karp improved scheduling algorithm.

7.1 XSU protocol

In this section, we evaluate the effect of our XSU protocol to eliminate the Hyper-Threading corruption erratum through a set of before and after examples. All examples in this section are run on an Intel Core i7-4770 processor with Hyper-Threading enabled. For all the examples, we are using the triad program which is a single-threaded, very stable workload performing loads and stores. We pin one instance on each of the sibling threads and we let these instances run "forever". The sibling threads we are measuring on are CPU0 (also referred to as HT_0) and CPU4 (also referred to as HT_1). We first compare with the examples showed in section 4.1.

In the first example, we measure the corrupting event MEM_UOPS_RETIRED: ALL_LOADS event (code 0x81d0) and the non-corrupting event MISPREDICTED_ BRANCH_RETIRED event (code 0x00c5). As we have explained, we expect the value of the branch misprediction event to be low, since it does not occur in tight loops. Indeed, if we only measure the branch mispredictions on CPU4 we get a value of 772 as the total count.

```
$ taskset -c 0 triad &
$ perf stat -r 10  -a -C 4 -e r00c5:u taskset -c 4 triad

  772 r00c5:u
```

Then, we add the measurement of the corrupting retired loads event on CPU0 (sibling thread) with the XSU protocol disabled:

```
# echo 0 >/sys/devices/cpu/ht_bug_workaround
```

```
$ taskset -c 0 triad &
$ taskset -c 4 triad &
$ perf  stat -a -C 0 -e r81d0:u &
$ perf stat -r10 -a -C 4 -e r00c5:u

  40 581 657 r00c5:u
```

We see the huge corruption of the branch misprediction event whose value is 40 581 657 instead of 772. Finally, we enable the XSU protocol and we try the same measurement:

```
# echo 1 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ taskset -c 4 triad &
$ perf  stat -a -C 0 -e r81d0:u &
$ perf stat -r10 -a -C 4 -e r00c5:u

  781 r00c5:u
```

The output is now back to expected, at 781. We manage to get the correct result because the XSU protocol scheduled the 0x81d0 and 0x00c5 events on different counters on each sibling thread, taking into account the dynamic constraints, not just the static ones as it was happening before. The valid configuration of the counters is shown in Fig. 7.1.

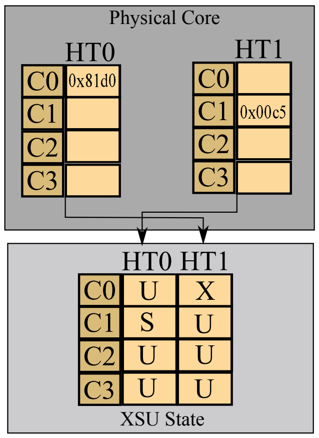


Figure 7.1: XSU corrected assignment for 0x81d0 (C) and 0x00c5 (NC) events

If we measure just enough events in a single run in order not to have multiplexing, then there is a case where we can avoid corruption even without enabling the XSU protocol. This case relies on the fact that the same event list is scheduled on both sibling threads at exactly the same time and that the list is not modified during the run. The two event lists are then synchronized and across sibling counters the same events are scheduled. This way, non-corrupting events cannot get corrupted by the corrupting ones. However, this implies that no other tool is monitoring the same process or CPU at the same time. This may be valid on single user systems, but not on shared servers. The XSU protocol eliminates the risk of corruption should the event lists on the sibling threads differ or run asynchronously, but it comes at the price of extra multiplexing as we demonstrate in the next example.

We measure 3 events, 2 corrupting (0x81d0, 0x08d1) and 1 non-corrupting (0x20cc) as shown below:

0x81d0	MEM_UOPS_RETIRED:ALL_LOADS
0x08d1	MEM_LOAD_UOPS_RETIRED:L1_MISS
0x20cc	ROB_MISC_EVENTS:LBR_INSERTS

These events are statically unconstrained, i.e., they can be scheduled on any of the 4 generic counters, hence they should all fit without requiring multiplexing. The non-corrupting event is the ROB_MISC_EVENT.LBR_INSERTS (code 0x20cc) which counts the number of entries inserted in the Last Branch Record (LBR) buffer. We choose this event because, as the LBR is not used on the test system, its value must always be zero. Hence, the corruption is easily identifiable. Indeed, when we measure it alone on CPU4 for ten times (-r10), we get 0 as value with 0% deviation for all the ten runs.

```
$ perf stat -a -C4 -r10 -e r20cc:u taskset -c 4 triad

  0      r20cc:u      (+- 0,00\%)
```

Now, with XSU disabled, we measure the list of these three events, on both siblings, in a single, combined run, for 10 seconds.

```
# echo 0 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ taskset -c 4 triad &
$ perf stat -a -A -C0,4 -e r20cc:u,r81d0:u,r08d1:u sleep 10

  CPU0                    0      r20cc:u      [100,00%]
  CPU4                    0      r20cc:u      [100,00%]
  CPU0          5 747 793 793      r81d0:u      [100,00%]
  CPU4          5 711 985 901      r81d0:u      [100,00%]
  CPU0            305 944 783      r08d1:u      [100,00%]
  CPU4            305 472 307      r08d1:u      [100,00%]

  10,000860002 seconds time elapsed
```

As we explained there is no multiplexing (100% scaling factor) since in each thread there are three events to be measured on four counters. But also, there is no corruption even without XSU, because the event lists are completely aligned and they do not change during the run.

Now, if the event lists on the sibling threads are not identical or the measurements are not initiated at the same time, which is most commonly the case, we will obtain corrupted results.

Indeed, with XSU disabled, we now measure the events in parallel but separate runs on the sibling threads and we change the order of these events in the two lists.

```
# echo 0 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ taskset -c 4 triad &
$ perf stat -a -A -C0 -e r81d0:u,r08d1:u,r20cc:u sleep 10 &
$ perf stat -a -A -C4 -e r20cc:u,r81d0:u,r08d1:u sleep 10

  CPU0         5 609 439 398      r81d0:u      [100,00%]
  CPU0           304 559 413      r08d1:u      [100,00%]
  CPU0           137 640 089      r20cc:u      [100,00%]

  10,000851695 seconds time elapsed


  CPU4           137 641 709      r20cc:u      [100,00%]
  CPU4         5 713 420 846      r81d0:u      [100,00%]
  CPU4           166 553 900      r08d1:u      [100,00%]

  10,000824227 seconds time elapsed
```

The output shows that there is still no multiplexing but there is corruption. The LBR inserts event (0x20cc) of CPU0 is corrupted by 137 640 089 counts leaked from the L1 misses event (0x08d1) scheduled on the sibling counter of CPU4. Similarly, the LBR inserts event of CPU4 is corrupted by 137 641 709 counts leaked from the loads retired event (0x81d0) scheduled on the sibling counter of CPU0. The corruption is shown in Fig 7.2.

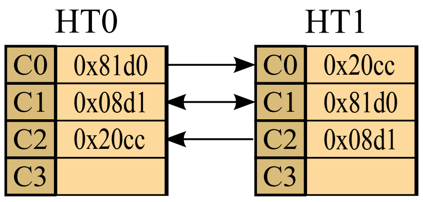


Figure 7.2: Corrupted measurements of 0x20cc, 0x81d0, 0x08d1

Next, we enable the XSU protocol and we run the same test.

```
# echo 1 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ taskset -c 4 triad &
$ perf stat -a -A -C0 -e r81d0:u,r08d1:u,r20cc:u sleep 10 &
$ perf stat -a -A -C4 -e r20cc:u,r81d0:u,r08d1:u sleep 10

  CPU0         5 650 303 673      r81d0:u      [100,00%]
  CPU0           166 632 879      r08d1:u      [100,00%]
  CPU0                     0      r20cc:u      [100,00%]

  10,000773541 seconds time elapsed


  CPU4                     0      r20cc:u      [66,68%]
  CPU4         5 509 435 203      r81d0:u      [66,66%]
```

```
CPU4          162 522 829      r08d1:u      [33,35%]

10,000779763 seconds time elapsed
```

With the XSU dynamic constraints, there is now multiplexing as we can see from the decreased fraction of time each event of CPU4 is measured on the hardware. The LBR inserts event value is back to zero for both threads. The XSU protocol protects the non-corrupting events from corruption. In Fig. 7.3, we show the configuration of the counters for the two threads, which also explains the scaling factors seen above.

As perf on CPU0 is initiated first, the events of CPU0 find all the counters in unused state and they are all scheduled in. As all events manage to get scheduled, there is no need of multiplexing and thus the scheduler of the generic layer will not schedule them out until the 10 seconds measurement is completed (Fig 7.3a). When CPU4 starts its measurements, some of the counters are already marked as exclusive or shared by CPU0. For the two corrupting events of CPU4 only C_3 is available while the non-corrupting event can use both C_2 and C_3. Hence, multiplexing will be needed and the generic scheduler will execute multiple passes rotating the event list as explained in chapter 3 At the first pass (Fig 7.3b), non-corrupting 0x20cc is the first event of the list and it is scheduled on C_2. The second event, the corrupting 0x81d0, is scheduled on C_3. The scheduler stops here because it fails scheduling the third event of the list, the corrupting 0x08d1, and the event list is rotated by one. At the second pass (Fig 7.3c), corrupting 0x81d0 is the first event of the list and it is scheduled on C_3. The scheduler fails to proceed because of lack of counters for the corrupting event 0x08d1 and both 0x08d1 and 0x20cc that follows, remain unscheduled. The event list is rotated by one. At the third pass (Fig 7.3d), corrupting 0x08d1 is at the head of the list and gets the opportunity to be scheduled on C_3. The second event, the non-corrupting 0x20cc, is scheduled on C_2. The scheduler fails to schedule the corrupting 0x81d0, the list is rotated and we return to the configuration of the first pass. Each pass lasts for a duration of a timer tick, which is by default 1 millisecond in the system we are testing on. For our 10 seconds measurement, the scheduler will do 10 000 passes. As we saw in the analysis above and the respective figures, every 3 passes the events 0x20cc and 0x81d0 are scheduled twice and the event 0x08d1 is scheduled once. Hence, the scaling factors 66.68% for 0x20cc and 66.66% for 0x81d0, which show that the events run on the hardware for 2/3rds of the time, are justified. Similarly, the scaling factor 33.35% for 0x08d1 shows correctly that this event is activated for 1/3rd of the time.

During the multiple passes of the scheduling algorithm, XSU guarantees valid assignment of the events on the counters. However, as it can be seen from the counts, the memory events have very different counts. Especially, the L1 misses event (0x08d1) drops from 305 944 783 to 166 632 879 on CPU0 and from 305 472 307 to 162 522 829 on CPU4, losing approximately 50% of its counts. We will

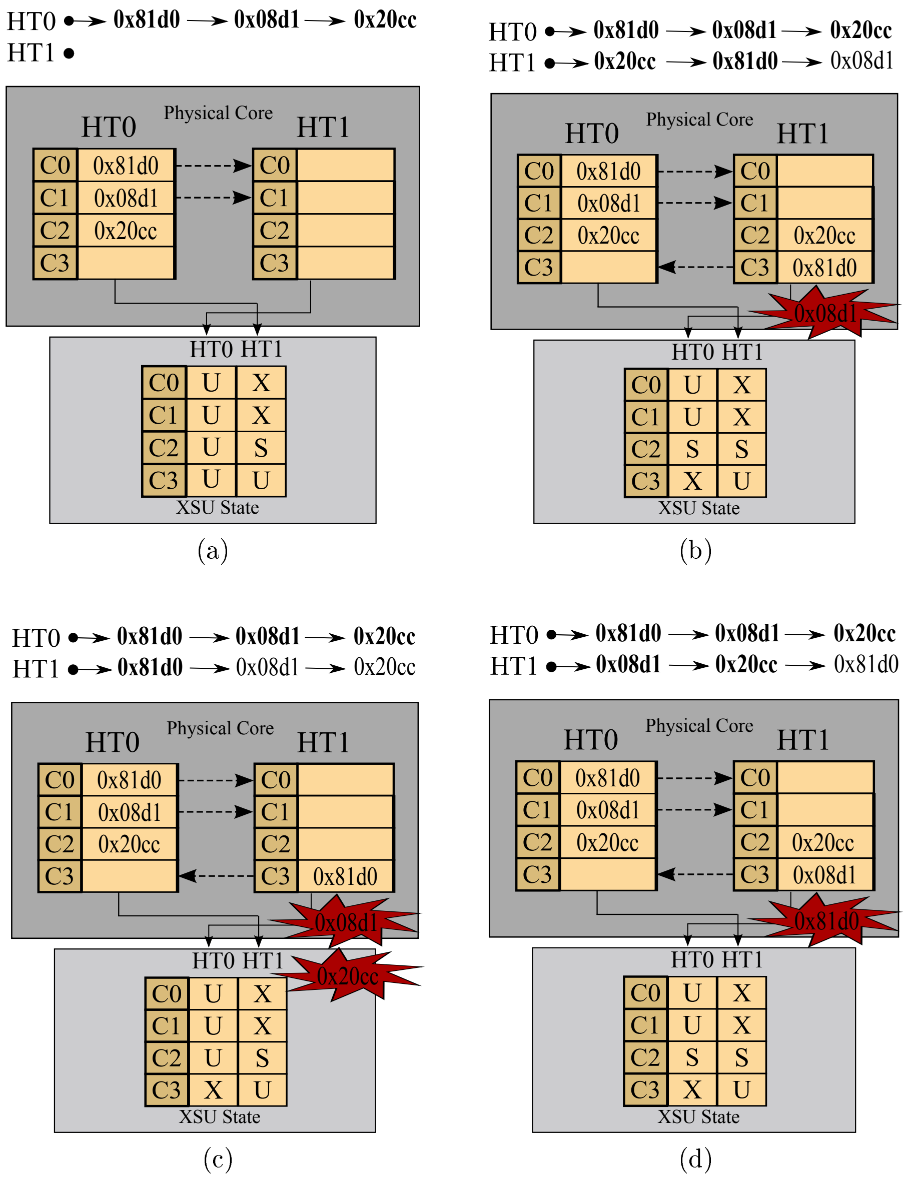


Figure 7.3: Valid measurements of 0x20cc, 0x81d0, 0x08d1 with XSU

explain this change later in chapter 8.

Now, let us examine cases where there are more events than counters including corrupting and non-corrupting events, without and with the XSU protocol.

In this first case, we are using 1 corrupting memory event (0x08d1) and 4 non-corrupting events (0x02c4, 0x20cc, 0x00c4, 0x010e) as shown below:

0x08d1	MEM_LOAD_UOPS_RETIRED:L1_MISS
0x02c4	BR_INST_RETIRED:NEAR_CALL
0x20cc	ROB_MISC_EVENTS:LBR_INSERTS
0x10c4	BR_INST_RETIRED:NOT_TAKEN
0x08c4	BR_INST_RETIRED:NEAR_RETURN

We have our triad workload running on both sibling threads (CPU0, CPU4). Because we will incur multiplexing and that will likely cause corruption, we first measure the non-corrupting events by themselves to get the actual counts in a single run with XSU disabled:

```
# echo 0 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ perf stat -r10 -a -C0 -e r02c4:u,r20cc:u,r10c4:u,r08c4:u sleep 10

  Performance counter stats for 'system wide' (10 runs):

    342      r02c4:u      (+- 0,04%)      [100,00%]
      0      r20cc:u      (+- 0,00%)      [100,00%]
  1 025      r10c4:u      (+- 0,04%)      [100,00%]
    342      r08c4:u      (+- 0,04%)      [100,00%]
```

As we can see, these are very stable events as shown by the deviation percentage over the 10 runs. These are our baseline numbers for the non-corrupting events obtained with no multiplexing. It is interesting to note that the values of the events BR_INST_RETIRED:NEAR_CALL (code 0x02c4) and BR_INST_RETIRED: NEAR_RETURN (code 0x08c4) in both threads are the same because they count the number of function calls and function returns respectively. Next, we add the corrupting memory event and we run the measurements of the five events with XSU disabled:

```
# echo 0 > /sys/devices/cpu/ht_bug_workaround
$ perf stat  -a -C4 -e r81d0:u,r02c4:u,r20cc:u,r10c4:u,r08c4:u sleep 10 &
$ perf stat  -a -C0 -e r81d0:u,r02c4:u,r20cc:u,r10c4:u,r08c4:u sleep 10

  5 587 605 755      r81d0:u      [80,00%]
     77 512 989      r02c4:u      [80,00%]
      6 087 383      r20cc:u      [80,00%]
      6 080 780      r10c4:u      [80,00%]
     18 218 197      r08c4:u      [79,99%]

  10,000809054 seconds time elapsed


  5 585 007 977      r81d0:u      [80,00%]
     18 233 527      r02c4:u      [80,00%]
      6 104 578      r20cc:u      [80,00%]
      6 091 492      r10c4:u      [80,00%]
     77 517 725      r08c4:u      [79,99%]
  10,000739612 seconds time elapsed
```

As expected, there is multiplexing. The scaling factor is about 80% which indicates that each event is active about 4/5th of the time, i.e., at any one time 4 out of 5 events are active on the hardware. Because of multiplexing, the event lists on both threads rotate at each timer tick. Thus, across the 10 seconds measurements, the corrupting event of the one thread ends up in facing and corrupting each one of the non-corrupting events of the other thread and vice-versa. All the non-corrupting events of both sibling threads have their results corrupted by several million of leaked counts. The more time a non-corrupting event finds itself scheduled with a corrupting event on a pair of sibling counters, the more unreasonably higher its value becomes due to corruption. The errors are prominent on these low frequency non-corrupting events we have chosen.

Next, we run the same test case but with XSU enabled and look at the impact on the counts of the non-corrupting events and on the multiplexing.

```
# echo 1 > /sys/devices/cpu/ht_bug_workaround
$ perf stat  -a -C4 -e r81d0:u,r02c4:u,r20cc:u,r10c4:u,r08c4:u sleep 10 &
$ perf stat  -a -C0 -e r81d0:u,r02c4:u,r20cc:u,r10c4:u,r08c4:u sleep 10

  5 554 095 990      r81d0:u      [30,99%]
            342      r02c4:u      [50,60%]
              0      r20cc:u      [60,40%]
          1 021      r10c4:u      [70,20%]
            350      r08c4:u      [70,17%]

  10,000809047 seconds time elapsed


  5 543 881 332      r81d0:u      [40,77%]
            336      r02c4:u      [50,59%]
              0      r20cc:u      [60,39%]
          1 030      r10c4:u      [70,19%]
            343      r08c4:u      [79,99%]

  10,000782512 seconds time elapsed
```

The counts have returned back their expected values according to the baseline numbers we collected at the beginning of this test. The other interesting part of these results is the scaling factor which has decreased, denoting a smaller active time on the hardware for each event. The difference between the values of the events BR_INST_RETIRED:NEAR_CALL (code 0x02c4) and BR_INST_RETIRED: NEAR_RETURN (code 0x08c4) in both threads, is due to multiplexing inaccuracies. The pattern of function calls and function returns may present changing phases which cannot always be captured because the events are not measured at all times. In any case, the values of these two events are very similar and the discrepancies of less than 10 counts we obtain now is far from the discrepancies of 60 000 000 we were obtaining when measuring without the XSU protocol.

In this example the sibling threads are competing to acquire the shared state and schedule their events on the counters every timer tick, because they are both subject to multiplexing. Hence, the time that each event of the two threads is actually ran on the hardware and thus the scaling factors of each event, cannot be predicted because of these race conditions. What the XSU protocol offers is a guarantee of valid configuration of the counters at all times and correct results for the non-corrupting events.

If we add more events to measure in a single run, we can see the effect of the XSU protocol on multiplexing. For this second case, we use an event list of 10 events on each thread, 4 corrupting memory events (0x81d0, 0x08d1, 0x10d1, 0x01d1) and 6 non-corrupting events (0x02c4, 0x20cc, 0x10c4, 0x08c4, 0x01c9, 0x04c8), as shown below:

0x81d0	MEM_UOPS_RETIRED:ALL_LOADS
0x08d1	MEM_LOAD_UOPS_RETIRED:L1_MISS
0x10d1	MEM_LOAD_UOPS_RETIRED:L2_MISS
0x01d1	MEM_LOAD_UOPS_RETIRED:L1_HIT
0x02c4	BR_INST_RETIRED:NEAR_CALL
0x20cc	ROB_MISC_EVENTS:LBR_INSERTS
0x10c4	BR_INST_RETIRED:NOT_TAKEN
0x08c4	BR_INST_RETIRED:NEAR_RETURN
0x01c9	RTM_RETIRED:START
0x04c8	HLE_RETIRED:ABORTED

The run without XSU will incur multiplexing, so we need to measure the actual counts of the events. We have the baseline numbers for the first 4 non-corrupting events (0x02c4, 0x20cc, 0x10c4, 0x08c4) from the previous example. We now measure the actual values of the 2 new non-corrupting events (0x01c9, 0x0408) in a single run with XSU disabled:

```
# echo 0 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ perf stat -r10 -a -C0 -e r02c4:u,r20cc:u,r10c4:u,r08c4:u sleep 10

  Performance counter stats for 'system wide' (10 runs):

  0       r01c9:u       (+- 0,00%)       [100,00%]
  0       r04c8:u       (+- 0,00%)       [100,00%]
```

The RTM_RETIRED:START and the HLE_RETIRED:ABORTED are events related to transactional memory support in Haswell processors. The RTM_RETIRED:START counts the number of times the restricted transactional memory execution starts and the HLE_RETIRED:ABORTED counts the number of aborted hardware lock elison transactions. The workload does not use transactional memory, so these events should have zero counts.

We disable the XSU protocol and we run the measurements of these 10 events on both sibling threads for 10 seconds.

```
# echo 0 > /sys/devices/cpu/ht_bug_workaround
$ perf stat  -a -C4 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u,r02c4:u, \
                      r20cc:u,r10c4:u,r08c4:u,r01c9:u,r04c8:u sleep 10 &
$ perf stat  -a -C0 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u,r02c4:u,r20cc:u, \
                      r10c4:u,r08c4:u,r01c9:u,r04c8:u sleep 10

  5 580 691 917      r81d0:u      [40,00%]
    196 376 492      r08d1:u      [40,01%]
    204 266 718      r10d1:u      [40,01%]
  3 112 470 528      r01d1:u      [40,01%]
     74 285 531      r02c4:u      [40,00%]
      8 669 406      r20cc:u      [40,00%]
      8 603 632      r10c4:u      [40,00%]
         12 409      r08c4:u      [40,00%]
     13 341 193      r01c9:u      [40,00%]
     17 087 329      r04c8:u      [39,99%]

  10,000915604 seconds time elapsed


  5 633 519 354      r81d0:u      [40,00%]
    187 004 134      r08d1:u      [40,00%]
    133 622 500      r10d1:u      [40,00%]
  3 014 317 138      r01d1:u      [40,01%]
     10 734 813      r02c4:u      [40,00%]
      3 792 868      r20cc:u      [40,00%]
      6 031 812      r10c4:u      [40,00%]
     16 528 567      r08c4:u      [40,00%]
     83 827 911      r01c9:u      [40,00%]
    100 139 343      r04c8:u      [39,99%]

  10,000943248 seconds time elapsed
```

As we can see, the values of all non-corrupting events have been severely impacted by the corrupting ones and the results we obtain are completely irrelevant with the actual behavior of our workload. We now enable the XSU protocol and repeat the measurements.

```
# echo 1 > /sys/devices/cpu/ht_bug_workaround
$ perf stat -a -A -C4 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u,r02c4:u, \
                      r20cc:u,r10c4:u,r08c4:u,r01c9:u,r04c8:u sleep 10 &
$ perf stat -a -A -C0 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u,r02c4:u,r20cc:u, \
                      r10c4:u,r08c4:u,r01c9:u,r04c8:u sleep 10

  CPU4      5 491 248 074      r81d0:u      [1,61%]
  CPU4        161 579 927      r08d1:u      [1,62%]
  CPU4        108 975 096      r10d1:u      [1,63%]
  CPU4      2 962 988 436      r01d1:u      [1,64%]
  CPU4                436      r02c4:u      [11,24%]
  CPU4                  0      r20cc:u      [20,82%]
  CPU4              1 056      r10c4:u      [30,39%]
  CPU4                323      r08c4:u      [30,36%]
```

```
CPU4                  0      r01c9:u      [39,93%]
CPU4                  0      r04c8:u      [39,94%]

10,000895995 seconds time elapsed


CPU0      5 518 336 748      r81d0:u      [39,99%]
CPU0        162 876 379      r08d1:u      [39,98%]
CPU0        109 572 002      r10d1:u      [39,98%]
CPU0      2 965 128 722      r01d1:u      [39,97%]
CPU0                328      r02c4:u      [39,97%]
CPU0                  0      r20cc:u      [39,98%]
CPU0                990      r10c4:u      [39,99%]
CPU0                328      r08c4:u      [40,00%]
CPU0                  0      r01c9:u      [39,99%]
CPU0                  0      r04c8:u      [40,00%]

10,000943248 seconds time elapsed
```

The results have returned to their expected levels but multiplexing has significantly increased especially on CPU4. Obviously, CPU0 is the winner of the race conditions between the sibling threads for accessing the XSU shared state and scheduling the events. CPU0 manages to keep the scaling factors at the same levels as without XSU. On the other hand, the fraction of time the events of CPU4 are scheduled on the hardware has decrease. The decrease is most severe for the corrupting events where the scaling factor drops below 2%. This is reasonable since these events can only be measured on unused counters, something very difficult to find given that CPU0 wins most of the race conditions and has its events scheduled first. However, the inaccuracies in the results of the corrupting memory events of CPU4 are not significant because, as we have explained, the workload has a very stable behavior with respect to memory operations. Again, we observe discrepancies between the function call event (code 0x02c4) and the function return event (code 0x08c4) on CPU4 but this can be attributed to the different scaling factors combined with the phases of the function call and return pattern. On the other hand, on CPU0 the scaling factors are the same and the values of 0x02c4 an 0x08c4 are also the same.

7.2 Scheduling optimization

To test various scheduling optimizations, we have developed a event scheduling simulator. With such a tool, we can more easily experiment with scheduling options without having to recompile a kernel and reboot the machine (real or virtual).

The simulator is a C program which includes the verbatim code from the perf_events subsystem with a shim layer to glue with standard user level code. The simulator includes the constraint tables for Intel and AMD X86 processors. It can be exercised with actual event names because it is linked with the libpfm4 [30] open-source library which provides event tables for all processors. Events can easily be listed with their static constraints as shown below:

```
----------------------------------------
Processor Name            : Sandy Bridge
Generic Counters No       : 4
Fixed-Purpose Counters No : 3
----------------------------------------
#--------------#----------------#--------------------------------
#    CODE      #  CONSTRAINTS   # NAME
#--------------#----------------#--------------------------------
0x0000000001b6 | 0x00000000000f | AGU_BYPASS_CANCEL:COUNT
0x000000000114 | 0x00000000000f | ARITH:FPU_DIV_ACTIVE
0x000001040114 | 0x00000000000f | ARITH:FPU_DIV
0x000000001fe6 | 0x00000000000f | BACLEARS:ANY
0x000000004188 | 0x00000000000f | BR_INST_EXEC:NONTAKEN_COND
0x000000008188 | 0x00000000000f | BR_INST_EXEC:TAKEN_COND
0x000000008288 | 0x00000000000f | BR_INST_EXEC:TAKEN_DIRECT_JUMP
```

The simulator includes multiplexing support, though it is not timed-based but simply based on the number of maximum scheduling iterations specified by the user. Once an iteration of the scheduling is done, it is followed by another until this number is reached. In case multiplexing is required, the event list is rotated by one event before the next scheduling iteration, emulating the behavior of the perf_events generic layer described in chapter 3. Each iteration can be dumped by the simulator for inspection. Below we demonstrate a simple example, using the standard scheduling algorithm (default) for two events (-e option) and measuring for 10 iterations (-n option). The counter assignment shown before every event name at each iteration is the counter on which the event was scheduled the last time it ran. In the example below we have cntr0 for uops_retired:any event and and cntr1 for mispredicted_branch_retired event. The number in front of the counter index is the cumulative percentage of time the event was scheduled up to the iterations shown, i.e., the scaling factor we were obtaining in our command line examples of the previous section. Here 100% means that both event were scheduled at each one of the 10 iterations.

```
$ sched_sim -n 10 -e uops_retired:any, mispredicted_branch_retired
----------------------------------------
Processor Name            : Haswell
Generic Counters No       : 4
Fixed-Purpose Counters No : 3
----------------------------------------
Event List:
1 - ---- ---- uops_retired:any            (code=0x5301c2, constraint=0xf)
2 - ---- ---- mispredicted_branch_retired (code=0x5300c5, constraint=0xf)

...
Iteration 10
1 - 100.00% cntr0 uops_retired:any            (constraint=0xf)
2 - 100.00% cntr1 mispredicted_branch_retired (constraint=0xf)
```

If events compete for counters, multiplexing is triggered and the percentages show the degree of multiplexing:

```
$ sched_sim -n 1000 -e l2_lines_in:any, \
                       l1d_pend_miss:occurrences,\
                       cycle_activity:stalls_l1d_pending
----------------------------------------
Processor Name            : Haswell
Generic Counters No       : 4
Fixed-Purpose Counters No : 3
----------------------------------------
Event List:
1 - ---- ---- l2_lines_in:any (code=0x5307f1, constraint=0xf)
2 - ---- ---- l1d_pend_miss:occurrences (code=0x0x1570148, constraint=0x4)
3 - ---- ---- cycle_activity:stalls_l1d_pending (code=0x85308a3, constraint=0x4)

...
Iteration 1000
scheduled 2 out of 3 events
1 + 66.70% cntr0 l2_lines_in:any (constraint=0xf)
2 + 66.70% cntr2 l1d_pend_miss:occurrences (constraint=0x4)
3 - 33.30% cntr2 cycle_activity:stalls_l1d_pending (constraint=0x4)
```

In the above example, l1d_pend_miss:occurrences and cycle_activity:stalls_ l1d_pending have constraint 0x4 which means they compete for counter 2. Only 2 out of 3 events can be scheduled at each iteration because of event list rotation and multiplexing. The + sign at an event row denotes that the event is scheduled on the current iteration while the - sign denotes that the event failed to be scheduled. The scaling factor reflects the multiplexing: 2/3rds of the time for l2_lines_in:any and l1d_pend_miss:occurrences and 1/3rd of the time for cycle_activity:stalls_l1d_ pending.

The Hopcroft-Karp scheduling algorithm is implemented in the simulator along with the greedy, first-match approach scheduling algorithm currently existing in the kernel. The algorithm used by the event scheduler can be selected from the command line.

It is also possible to experiment with the event constraints. Constraints masks can be passed from the command line to help with simulating more constrained environments such as when the XSU protocol is enabled. Below, we have an example with measuring events for which we have provided their dynamic constraints using the -C option. The order of the constraints is respective to the order in which the events are given in the event list. In the example below, l2_lines_in:any has constraint 0x6 and thus it supports counter 1 and counter 2, l1d_pend_miss:occurrences has constraint 0x8 and supports only counter 3, cycle_activity_stalls_l1d_pending has constraint 0x9 and supports counter 0 and counter 3 and finally inst_retired_any_p event has constraint 0xb and supports counter 0, counter 1 and counter3 (-C 0x6,0x8,0x9,0xb).

We first run this simulation with the default, greedy scheduling algorithm:

```
$ sched_sim -n 1000 -e l2_lines_in:any,
                       l1d_pend_miss:occurrences,\
                       cycle_activity_stalls_l1d_pending,\
                       inst_retired_any_p\
                    -C 0x6,0x8,0x9,0xb
----------------------------------------
Processor Name            : Haswell
```

```
Generic Counters No       : 4
Fixed-Purpose Counters No : 3
----------------------------------------
Event List:
1 - ---- ---- l2_lines_in:any (code=0x5307f1, constraint=0x6)
2 - ---- ---- l1d_pend_miss:occurrences (code=0x1570148, constraint=0x8)
3 - ---- ---- cycle_activity:stalls_l1d_pending (code=0x85308a3, constraint=0x9)
4 - ---- ---- inst_retired:any_p (code=0x5300c0, constraint=0xb)

...
Iteration 1000
Scheduled 3 out of 4 events
1 + 75.00 cntr-0 l2_lines_in:any (constraint=0x6)
2 + 75.00 cntr-1 l1d_pend_miss:occurrences (constraint=0x8)
3 + 75.00 cntr-3 cycle_activity:stalls_l1d_pending (constraint=0x9)
4 - 00.00 cntr-0 inst_retired:any_p (constraint=0xb)
```

We can run the same scheduling instance using the Hopcroft-Karp algorithm (-M option) and compare the results:

```
$ sched_sim -M -n 1000 -e l2_lines_in:any,\
                          l1d_pend_miss:occurrences,\
                          cycle_activity_stalls_l1d_pending,\
                          inst_retired_any_p\
                       -C 0x6,0x8,0x9,0xb
----------------------------------------
Processor Name            : Haswell
Generic Counters No       : 4
Fixed-Purpose Counters No : 3
----------------------------------------
Event List:
1 - ---- ---- l2_lines_in:any (code=0x0x5307f1, constraint=0x6)
2 - ---- ---- l1d_pend_miss:occurrences (code=0x1570148, constraint=0x8)
3 - ---- ---- cycle_activity:stalls_l1d_pending (code=0x85308a3, constraint=0x9)
4 - ---- ---- inst_retired:any_p (code=0x5300c0, constraint=0xb)

...
Iteration 1000
scheduled 4 out of 4 events
1 + 100.00 cntr-2 l2_lines_in:any (constraint=0x6)
2 + 100.00 cntr-3 l1d_pend_miss:occurrences (constraint=0x8)
3 + 100.00 cntr-0 cycle_activity:stalls_l1d_pending (constraint=0x9)
4 + 100.00 cntr-1 inst_retired:any_p (constraint=0xb)
```

The goal of the optimal scheduling algorithm we have implemented is to maximize the use of the counters while respecting the constraints. In the comparison above, we see that the scaling factor of the events using Hopcroft-Karp algorithm is maxed out at 100%, which means that all the events were scheduled at each iteration. At the same scheduling instance the standard, greedy algorithm yields a 75% scaling factor meaning it could only schedule 3 out of 4 events at each iteration. Thus, for this constraint configuration, the Hopcroft-Karp algorithm fares much better. The PMU is more utilized, resulting in no multiplexing and increased accuracy.

In order to evaluate the improvement that Hopcroft-Karp brings, we generalize the test case. We run all possible constraint configurations of 4 events to 4 generic counters from -C 0xf,0xf,0xf,0xf to -C 0x1,0x1,0x1,0x1, i.e. $15^4 = 50625$ scheduling instances and we get a full evaluation of each scheduling algorithm. By being optimal, the Hopcroft-Karp algorithm is either giving the same scheduling with the existing, greedy algorithm or it is performing better, i.e., it achieves more events to be scheduled on the counters. The Hopcroft-Karp algorithm gives better scheduling to 5950 instances, achieving approximately 12% improvement on counter utilization and thus on measurement accuracy.

With the simulator we have implemented, it is also possible to modify the way the generic perf_events layer is incrementally passing events to the x86 architecture specific layer. For instance, it is possible to continue passing events continuing beyond the event for which the first error occurred. For N generic counters we have experimented passing scheduling instances with $2 * N$ events regardless on which event the first failure occurs. With such small change, we have demonstrated that the Hopcroft-Karp algorithm can provide an increased utilization of the counters and a better measurement accuracy in up to 18% of the cases, while the effect on the execution time is negligible. In the future, we intend to modify the perf_events generic layer in the kernel to continue after the first error for a number of events which is a polynomial expression of the number of generic counters.

Chapter 8

Future Work

In the previous examples we have seen that the values of the corrupting events change between the single-thread measurements, the runs without the XSU and the runs with XSU. In the following example we focus on the measurements of the four corrupting events we encountered so far and we explain how their counts can differ throughout the tests.

0x81d0	MEM_UOPS_RETIRED:ALL_LOADS
0x08d1	MEM_LOAD_UOPS_RETIRED:L1_MISS
0x10d1	MEM_LOAD_UOPS_RETIRED:L2_MISS
0x01d1	MEM_LOAD_UOPS_RETIRED:L1_HIT

First, we measure these events on both siblings, in a single, combined run, with XSU disabled.

```
# echo 0 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ taskset -c 4 triad &
$ perf stat -a -A -C0 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u sleep 10 &
$ perf stat -a -A -C4 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u sleep 10

  CPU0      5 565 857 411      r81d0:u      [100,00%]
  CPU0        292 858 367      r08d1:u      [100,00%]
  CPU0        213 411 639      r10d1:u      [100,00%]
  CPU0      2 908 809 947      r01d1:u      [100,00%]

  10,004670257 seconds time elapsed

  CPU4      5 605 756 134      r81d0:u      [100,00%]
  CPU4        292 211 911      r08d1:u      [100,00%]
  CPU4        213 045 877      r10d1:u      [100,00%]
  CPU4      2 938 263 812      r01d1:u      [100,00%]

  10,004583900 seconds time elapsed
```

All the events fit in the 4 generic counters, thus there is no multiplexing (100% scaling factor). With XSU protocol disabled the configuration of the counters is shown in Fig. 8.1 below and there is corruption between the memory events scheduled on the same counter.

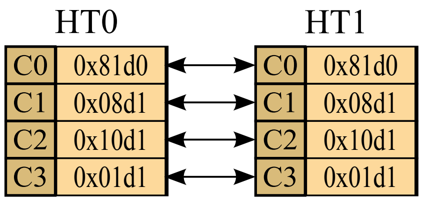


Figure 8.1: Measuring 0x81d0, 0x08d1, 0x10d1, 0x01d1 with XSU disabled

If we rerun the test with XSU enabled we get the following results.

```
# echo 1 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ taskset -c 4 triad &
$ perf stat -a -A -C0 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u sleep 10 &
$ perf stat -a -A -C4 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u sleep 10

  CPU0      5 465 161 580      r81d0:u      [50,01%]
  CPU0        157 542 131      r08d1:u      [50,00%]
  CPU0        104 608 156      r10d1:u      [49,99%]
  CPU0      2 879 525 359      r01d1:u      [50,00%]

  10,004670257 seconds time elapsed

  CPU4      5 418 290 816      r81d0:u      [50,01%]
  CPU4        156 484 117      r08d1:u      [50,00%]
  CPU4        103 779 978      r10d1:u      [49,99%]
  CPU4      2 918 604 417      r01d1:u      [50,00%]

  10,004583900 seconds time elapsed
```

The configuration of the counters with XSU protocol is shown in Fig 8.2 below.

We observe that the values of the memory events have significantly dropped. Especially, the L1 misses event (0x08d1) and the L2 misses event (0x10d1) have lost more than 46% and 51% of their counts respectively. The memory loads event (0x81d0) and the L1 hits event (0x01d1) are high-frequency events and their percent losses are lower but still important. This is explained by the fact that these event are leaking their counts on the unused sibling counters and these leaked counts are not taken into account for the final value. In Fig. 8.2 above, this leak is represented as a dashed-line. This loss of counts did not appear when we measured with XSU disabled. At this case, the events measured on sibling counters were identical and the outcoming leaked counts of one were compensated by a similar number of incoming leaked counts from the other. Thus, the leaked counts were aggregated on the sibling counter and reported. It needs to be noted that the loss of counts is not a side effect of the XSU protocol and this can be shown easily with the following test. With XSU disabled we run the measurements of the same events only on one thread so that the counters of the sibling thread are unused:

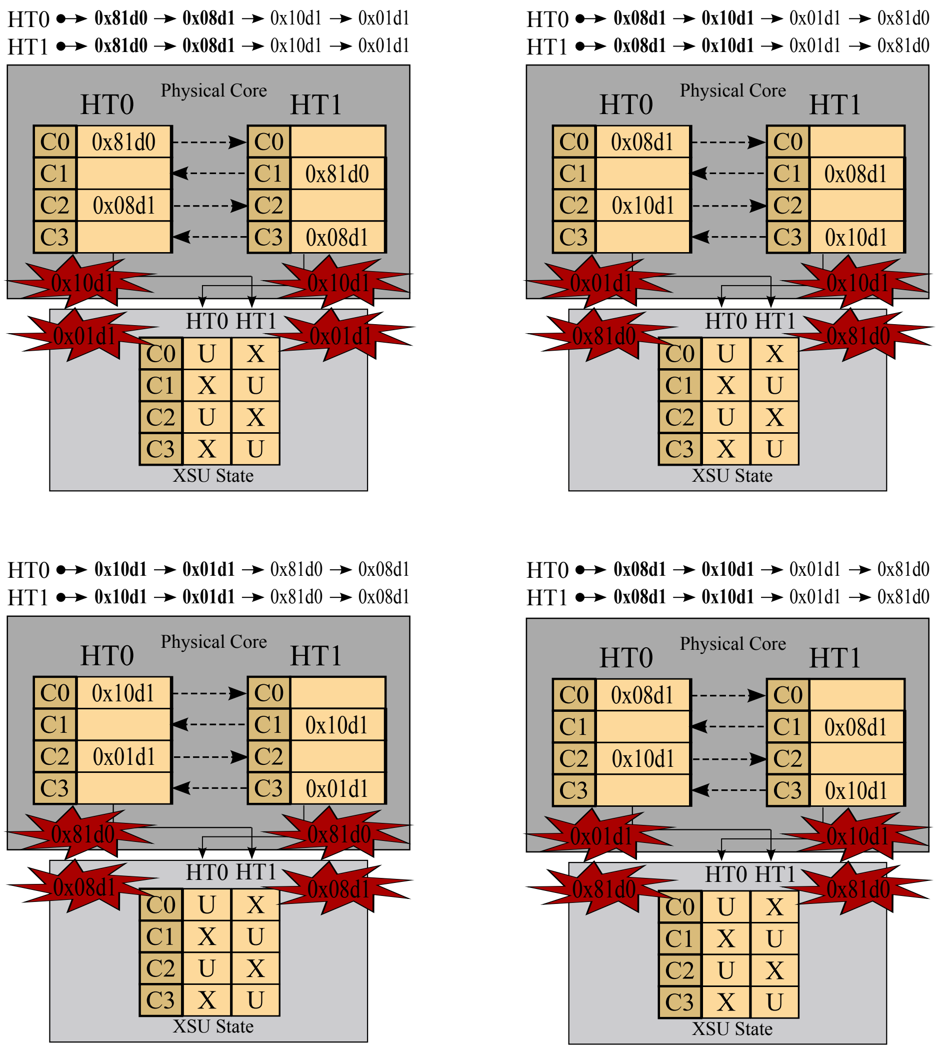


Figure 8.2: Measuring 0x81d0, 0x08d1, 0x10d1, 0x01d1 with XSU enabled

```
# echo 1 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ perf stat -a -A -C0 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u sleep 10 &

  CPU0      5 471 662 581      r81d0:u      [100,00%]
  CPU0        159 592 403      r08d1:u      [100,00%]
  CPU0        105 259 023      r10d1:u      [100,00%]
  CPU0      2 920 631 823      r01d1:u      [100,00%]

  10,004670257 seconds time elapsed
```

The configuration of the counters during this run is shown in Fig 8.3.

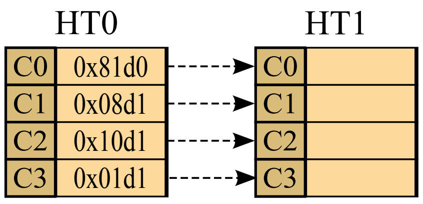


Figure 8.3: Measuring 0x81d0, 0x08d1, 0x10d1, 0x01d1 on one thread

In the output above we observe almost the same loss of counts as we did for the measurements on two threads with XSU enabled. Hence, the loss of counts is not caused by XSU but it is not prevented either. The primary goal of XSU is to protect non-corrupting events from getting corrupted by corrupting events when measured on sibling counters. The re-integration of the leaked counts in the final values of the corrupting events is not taken care of by XSU. However, XSU makes this re-integration simpler because when a corrupting event is measured on one counter, it is guaranteed by XSU that no valid event uses the sibling counter and thus all the counts accumulated there come from the measurement of the corrupting event. As future work, we intend to extend the XSU protocol in order to re-integrate the leaked counts in the final values of the corrupting events.

The XSU protocol addresses the PMU hardware erratum and successfully eliminates the counter corruption. However, the correctness comes at the cost of more constrained events, as shown in the examples above. The more constraints incurred by the XSU protocol combined with the race conditions between the sibling threads, may even cause an event of one thread to never get scheduled because of what is measured on the counters of the sibling thread. A simple artificial example is to measure on CPU0 our 4 known corrupting memory events for 10 seconds and on CPU4 the non-corrupting branches event for 5 seconds.

```
# echo 1 >/sys/devices/cpu/ht_bug_workaround
$ taskset -c 0 triad &
$ taskset -c 4 triad &
$ perf stat -a -A -C0 -e r81d0:u,r08d1:u,r10d1:u,r01d1:u sleep 10 &
$ perf stat -a -A -C4 -e branches sleep 5

  CPU0      5 470 671 639      r81d0:u      [100,00%]
  CPU0        157 698 323      r08d1:u      [100,00%]
  CPU0        106 859 230      r10d1:u      [100,00%]
  CPU0      2 918 823 504      r01d1:u      [100,00%]

  10,004326725 seconds time elapsed


  CPU4      <not counted>      branches

  5,004713819 seconds time elapsed
```

On CPU0, the 4 generic counters are used by corrupting events. On CPU4, only one event is measured yet it cannot get scheduled while the measurement runs on CPU0, even though it can statically run on any of the 4 generic counters. As per the XSU algorithm all 4 counters of CPU4 are in X state which means they cannot be used as shown in Fig. 8.4. No multiplexing is triggered on CPU0 because the number of events is equal to the number of counters and there is no error scheduling all the events at the first pass. This means that the events of CPU0 will not be scheduled out until the measurement of 10 seconds is over. Therefore, no event using the generic counters on CPU4 can be scheduled for this time period. This is why the cycles event is not counted.

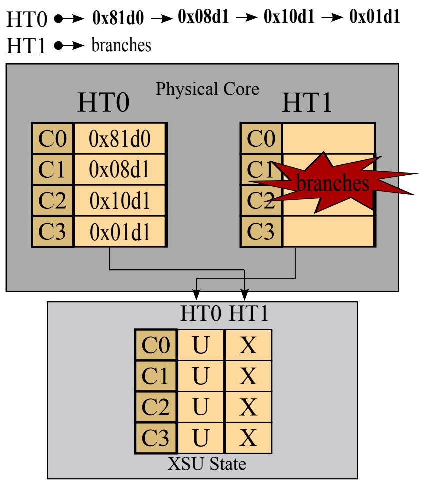


Figure 8.4: XSU state with 4 corrupting events on HT_0

It should be noted that this is not a weakness of the XSU protocol. To the contrary, it is an expected consequence of the more constrained environment required by the protocol in order to achieve correct results. Without this behavior, measurements, like the one of the example above, would yield corrupted counts.

However, we need ensure fairness between the two hyper-threads and give events on each thread's linked list a chance to access the counters, i.e., the hardware resource. No hyper-thread can starve the other one. We intend to address this issue in the future.

Chapter 9

Conclusions

In this thesis, we address several important issues related to hardware-based performance monitoring. Nowadays, all processors have a Performance Monitoring Unit (PMU) which provides a set of hardware counters to measure various micro-architectural events such as elapsed cycles or cache misses (c.f. Section 2.1). The PMU provides a unique insight into how software uses the underlying hardware resources. It is used count occurrences of micro-architectural events or collect statistical profiles with very low overhead to determine where there may be resource bottlenecks.

As demand for compute power increases constantly, the pressure on hardware resources rises. It is, therefore, important to making best use of available hardware resources. The scientific computing community, such as at CERN, has many physicists using PMU-based tools to improve the code analyzing data captured by Large Hadron Collider (LHC). In companies such as Google, PMU data help improve hardware utilization in data centers, code quality via feedback-directed compiler optimizations, hardware capacity planning and processor micro-architectural features.

The PMU is exposed to Linux users via the perf_events subsystem and system call (c.f. Section 2.2.1). This interface provides a large palette of features covering all the needs of performance analysts and developers. The event-based interface simplifies development of tools. Events may have constraints with regard to which PMU counters they can be measured and the event scheduling is at the core of the subsystem. The actual management of the PMU resource is handled by the kernel including how events are scheduled on counters (c.f. Section 3).

Recent Intel processors with Hyper-Threading support have a published erratum which may cause serious counter corruption across sibling hyper-threads, i.e. hyper-threads sharing the same physical core. Counters on one hyper-thread measuring certain corrupting events, leak their counts on sibling counters and corrupt their values (c.f. Section 4.1). The erratum impacts three generations of popular Intel processors: SandyBridge, IvyBridge and Haswell and there is no hardware or firmware solution to this problem. The corruption of the performance monitoring data makes any performance analysis unreliable and often misleading. This results in losing a valuable tool for understanding low-level performance problems and improving thousands of critical applications.

The first part of our effort has focused on developing a software workaround to eliminate the counter corruption. Mutual exclusion between counters across hyper-threads is enforced when corrupting events are used. Inspired by the MSI cache-coherence protocol, we have developed a sophisticated mechanism called XSU, which uses three states (eXclusive, Shared, Unused) required for any hyper-thread in order to distinguish which PMU counters can perform measurements without yielding corrupting results (c.f. Section 5). The integration of the protocol in leverages the perf_events event scheduling infrastructure.

Our XSU solution guarantees that all events can be measured safely. Corrupting events which are critical for any serious performance analysis need not to be banned to ensure correctness and non-corrupting events measurements can be trusted to reflect workload behavior. The protocol is implemented in the Linux kernel and there is no change to any user level tools. We have published the code to the Linux kernel community and our patches will be included in a future kernel releases soon.

The integration of XSU protocol in perf_events but the additional constraints derived the mutual exclusion requirements produce more constrained event scheduling instances. The current perf_events event scheduling algorithm uses a greedy, first-match approach which works very well when most events are unconstrained but the quality of its counter assignments degrades with XSU as events which were not constrained may become constrained because of events measured on the sibling hyper-thread.

In the second part of our effort, we have focused on improving the perf_events scheduling algorithm. We have first developed an event scheduling simulator in which we have imported the actual perf_events Intel and AMD x86 scheduling code from the kernel (c.f. Section 7.2). We use the simulator to experiment with scheduling algorithms, event constraints and their impact on multiplexing, i.e. the time-sharing of the PMU resource when all event constraints cannot be satisfied at once.

We have identified that the perf_events scheduling can be modeled as a matching problem in an unweighted bipartite graph. Hopcroft-Karp algorithm is a maximum cardinality matching algorithm for bipartite graphs which can be integrated in perf_events subsystem and provide optimal scheduling of events on counters with respect to the event constraints (c.f. Chapter 6). We have first implemented the Hopcroft-Karp algorithm in our simulator and have run several comparisons with the existing, greedy algorithm. Results show that for Intel's 4-counter configuration, the integration of Hopcroft-Karp algorithm in the Intel architecture specific layer of the perf_events subsystem improves counter utilization by 12%. Furthermore, we have also demonstrated that if the generic layer of the perf_events subsystem is modified slightly, the improvement can reach up to 18%. We have successfully implemented the complex Hopcroft-Karp algorithm in the Linux kernel following the stringent coding standards, including no recursion. The code exports a single entry point which can just be swapped with the existing call to the greedy algorithm. This code will eventually be contributed to the Linux kernel community.

Although, the XSU solution avoids the corruption, it does not produce valid counts for the corrupting events because their leaked counts are not re-integrated. We believe this could be fixed for counting mode events. Furthermore, we have shown that in certain conditions, some events may not be scheduled despite the integration of Hopcroft-Karp scheduling algorithm and the multiplexing support, because the PMU counters are unavailable due to events measured in the sibling thread. This issue could be solved at a higher level by ensuring more fairness between hyper-threads.

In summary, in this thesis, we have addressed two important issues related to the correctness and efficiency of hardware performance monitoring in the Linux kernel on Intel X86 processors. We have developed a sophisticated workaround for a serious cross hyper-thread counter corruption erratum guaranteeing measurement correctness and enabling usage of all events. With the help of the results from our PMU event scheduling simulator, we have implemented an alternative scheduling algorithm based on the Hopcroft-Karp maximum cardinality matching graph algorithm which can yield up to 18% better scheduling in the more constrained environment imposed by the XSU solution. We have contributed our code to the Linux kernel community and we have identified several possible extensions which we intend to address in the future. This work has received recognition from the performance monitoring community with an honorary award issued by Intel.

Bibliography

[1] *Intel 64 and IA-32 architecture software developer manual.* Intel Corporation, February 2014, vol. 3b.

[2] *BIOS and Kernel Developer's Guide for AMD Family 15h processors.* Advanced Micro Devices, January 2013, rev 3.14.

[3] *ARM Cortex-A15 MPCore Processor Technical Reference Manual.* ARM, 2013, ch. 11, rev 4p0.

[4] *PowerISA.* IBM, 2013, ch. 9, version 2.07.

[5] *Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide.* Intel, March 2012.

[6] *AMD GPU Performance API.* Advanced Micro Devices, 2014.

[7] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 379–391. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465388

[8] D. Chen and D. Li, "AutoFDO," https://www.youtube.com/watch?v=26SrOC6MXWg.

[9] "Intel VTUNE Amplifier XE," https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[10] D. Levinthal, "The gooda performance analysis tool," http://code.google.com/p/gooda.

[11] "Perf_events tutorial," http://perf.wiki.kernel.org/.

[12] V. Weaver, "Perf_events programming guide," http://web.eece.maine.edu/~weaver/projects/perf_events/programming.html.

[13] *Intel Hyper-Threading Technology: Technical User's Guide.* Intel Corporation, January 2003.

[14] *Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual.* Intel, February 2014.

[15] "OProfile," http://oprofile.sf.net/.

[16] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" in *ACM Transactions on Computer Systems*, 1997, pp. 1–14.

[17] "FreeBSD," http://freebsd.org/.

[18] "FreeBSD PmcTools Wiki," https://wiki.freebsd.org/PmcTools.

[19] "Oracle Solaris Studio," http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html.

[20] M. Chynoweth and R. Chabukswar, "PBA: Performance and Power Analysis utilizing Intel Performance Bottleneck Analyzer," https://sites.google.com/site/analysismethods/isca2013/program-1, June 2013, iSCA 2013, workshop on Analysis Methodologies and Tools, Tel Aviv, Israel.

[21] M. Chynoweth, "Utilizing Performance Bottleneck Analyzer to debug issues on Intel's future SOCs," http://indico.cern.ch/event/280897/, Nov 2013, 2nd CERN Advanced Performance Tuning Workshop.

[22] A. Yasin, "Top down analysis : Never lost with perf counters," https://sites.google.com/site/analysismethods/isca2013/program-1, June 2013, ISCA'13, workshop on Analysis Methodologies and Tools, Tel Aviv, Israel.

[23] ——, "Top down analysis : Never lost with perf counters," http://indico.cern.ch/event/280897/, Nov 2013, 2nd CERN Advanced Performance Tuning Workshop.

[24] *2nd Generation Intel Core Processor Family Desktop, Intel Pentium processor Family Desktop, Intel Celeron processor family desktop specification update.* Intel Corporation, December 2013.

[25] *Desktop 3rd Generation Intel Core Processor Family specification update.* Intel Corporation, March 2014.

[26] *Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium processor Family, and desktop Intel Celeron processor family specification update.* Intel Corporation, March 2014.

[27] *Intel 64 and IA-32 architecture software developer manual.* Intel Corporation, February 2014, vol. 3a.

[28] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture, A Hardware/Software Approach.* Morgan Kaufman, September 1998.

[29] M. Karpinski and W. Rytter, *Fast parallel algorithms for graph matching problems.* Oxford University Press, May 1998.

[30] "Libpfm4: a helper library for performance tools," http://perfmon2.sf.net/.