Sapienza
Università di Roma

Ph.D. program in

Computer Engineering

XXXIV cycle - January 2022

Methodologies and techniques for on-line exploitation of Performance Monitor Units in modern computing systems

Advisor
Prof. Francesco Quaglia
Dr. Alessandro Pellegrini

Candidate
Stefano Carnà

Co-Advisor
Dr. Alessandro Pellegrini

Reviewers
XXX
XXX

Don't be afraid to give up the good
to go for the great.

Abstract

We are living in a hot period in which the intense competition among chipmakers is turning into an explosion of new hardware-level features. Among the innovations, modern processors embed a hardware support able to identify the footprint on the hardware caused by the execution of software.

In this thesis, we propose novel techniques to leverage the Performance Monitor Units (PMUs) hardware facilities offered by modern off-the-shelf CPUs to gather hardware-level footprint data at runtime, and exploit this information on-the fly.

Our solutions aim to collect hardware-level information while minimizing the perturbation experienced by the software applications and foster its online analysis and exploitation.

Throughout this thesis, we show how our methodology can be employed to address different problems in different domains. In particular, we provide the operating system with a module to master this integrated hardware support in order to perform dynamic corrective operations in malware detection, CPU-scheduling consolidation and functional hardware-accelerated tasks. Moreover, working at the lowest software layer has the extra benefit of enforcing the transparency property of our approach, strengthening both the adaptability and the dependability of the whole solution.

Beyond demonstrating how PMUs can be exploited for security, performance and functional activities, in this thesis, we additionally present our profiling infrastructure, which implements versatile mechanisms for the exploitation of PMUs in modern contexts.

Acknowledgment

I would like to express my sincere and most profound appreciation to all the people who have been part of this long, immersive and challenging journey.

I owe my deepest gratitude to my thesis advisors, Prof. Francesco Quaglia and Prof. Alessandro Pellegrini, for their wisdom and guidance throughout this research project and the invaluable supervision, support and tutelage during the course of my PhD degree.

I would like to offer my special thanks to Prof. Bruno Ciciani for his continuous willingness, especially when dealing with bureaucratic matters.

My sincere thanks also go to the High-Performance and Dependable Computing System (HPDCS) group at Sapienza Università di Roma. I will never forget the sleepless nights we worked together before deadlines and all the fun we had while improving our multitasking *skills.*

I would thank my buddies, Andrea and Matteo. You softened the latest part of this journey, giving me a flawless dose of friendship and pushing me to overcome my limits and uncertainties.

I am deeply grateful to thank my family: my parents, sister, and brother for supporting me spiritually throughout writing this thesis and my life in general.

Last but not least, I would express my most profound thankfulness to Idabelle. We have started this long journey together, and still together, we are reaching the end. You have always been by my side, helping and encouraging me, especially during the most challenging days. Part of this achievement is yours, and I couldn't wish for a better person to share it with.

Stefano Carnà

Contents

1 Introduction **1**

2 Performance Monitor Units Background **6**
- 2.1 Intel's Precise Event-Based Sampling . . . 8
- 2.2 AMD's Instruction-Based Sampling . . . 9
- 2.3 Technical/Scientific Problems and Needs . . . 10
- 2.4 Literature Review . . . 12

3 Exploiting hardware beyond offline analysis **15**
- 3.1 Efficient Collection of PMU Data . . . 16
- 3.2 Discriminating the Profiling Domain . . . 29
- 3.3 Performance Data Buffering . . . 38
  - 3.3.1 System-wide Buffering . . . 39
  - 3.3.2 Per-thread Buffering . . . 42
  - 3.3.3 Per-core buffer . . . 44
  - 3.3.4 Accessing Profiling Data from User Space . . . 45
- 3.4 Experimental Assessment . . . 46
  - 3.4.1 Efficient Collection of PMU Data . . . 47
  - 3.4.2 Discriminating the Profiling Domain . . . 50
  - 3.4.3 Accessing Profiling Data from User Space . . . 54

4 Online activity tracing for system security **55**
- 4.1 Related Work . . . 58
- 4.2 Threat Model . . . 59
- 4.3 Detecting Side-Channel Attacks . . . 60
  - 4.3.1 Architectural Details and Preliminary Work Hypotheses 60
  - 4.3.2 Detection Metrics . . . 62
  - 4.3.3 Setting up the thresholds . . . 65
- 4.4 System-Wide Detection and Reference Implementation . . . 66
  - 4.4.1 Selected Monitoring Events and Strategy to Acquire HPC Data . . . 66
  - 4.4.2 Observation Windows . . . 68
  - 4.4.3 Suspecting Malicious Processes . . . 69
- 4.5 Mitigation Strategies . . . 70
  - 4.5.1 Side-channel Attack Mitigations . . . 71

4.5.2 Transient Execution Mitigations 71
4.6 Experimental Assessment 73
4.6.1 Experimental Setup 73
4.6.2 Stability of HPC Events 74
4.6.3 Accuracy of the System-Wide Detection Approach . . 75
4.6.4 Performance Assessment 79

5 Microarchitectural-driven application co-scheduling for system consolidation 85
5.1 Related Work 87
5.2 The Consolidator infrastructure 88
5.2.1 The HPC driver 89
5.2.2 The Profiler 91
5.2.3 Finding the best Co-Scheduling 92
5.3 Experimental Results 96

6 Accelerating PDES via hardware-assisted incremental checkpointing 102
6.1 Related Work 103
6.2 The Hardware-assisted Incremental Checkpointing Architecture 105
6.2.1 Hardware-assisted Memory Write Tracing 105
6.2.2 Managing Incremental Checkpoints 108
6.3 Experimental Results 110

7 Conclusions 114

List of Figures

3.1 The general structure of IDT. 22
3.2 Stack patching schema 32
3.3 General Organisation of the Non-Blocking Circular Queue. . 40
3.4 Possible Queue States. 41
3.5 Workload overhead comparison between IRQ- and NMI-based PMIs. 48
3.6 Samples generation comparison between IRQ- and NMI-based PMIs. 49
3.7 Efficiency between IRQ- and NMI-based PMIs. 50
3.8 Runtime overhead of context-switch intensive workloads with active system hooks. 51
3.9 Comparison among different approaches to provide per-process private data. 51
3.10 Comparison among different buffering policies to collect process profiling data. 52
3.11 Perf tool evaluation at different sampling periods and 8 hardware events monitored 53
3.12 Time to fully read kernel level data via mmap- and VFS-based solutions. 54

4.1 Overall detection architecture and components involved in memory access—hit paths are not shown. 61
4.2 Detection accuracy evaluation for different values of γ ($\alpha, \beta = 1$). $\mathcal{S}_1$ and $\mathcal{P}_4$ indicate the direct and the indirect attacks, respectively, while *OK* indicates normal processes. 78
4.3 Percentage of a 256-byte secret that an attack can correctly extract before its detection for different values of γ ($\alpha, \beta = 1$) and victim's read rates. 80
4.4 Performance Effects of the HPC-based Monitoring System on different Architectures (logscale on the x axis). 82
4.5 Performance Penalties by Mitigations on the i5-8250U. . . . 83

5.1 High-level representation of the tool structure and interaction among the internal components. 90

5.2 Consolidator process to find the partition that provides the best score. 93
5.3 Consolidator process to find the partition that provides the best score. Stress-ng test with 8 parallel workers per stressor. 98
5.4 Score of the different configurations of stress-ng workloads. . 99
5.5 Energy metric of the different configurations of the stress-ng workloads. 99
5.6 CPU occupancy metric of the different configurations of the stress-ng workloads. 99
5.7 Useful Work metric of the different configurations of the stress-ng workloads. 100
5.8 Score of the different configurations of the QEMU-based experiment. 100
5.9 Energy metric of the different configurations of the QEMU-based experiment. 101
5.10 CPU usage metric of the different configurations of the QEMU-based experiment. 101
5.11 Useful Work metric of the different configurations of the QEMU-based experiment. 101

6.1 Average Event Duration. 112
6.2 Simulation Efficiency. 112

List of Tables

4.1 An overview of considered cache side-channel attacks and references to the used implementations. 74
4.2 Comparison between HPCs and Software Instrumentation on all the architectures. Err represents the distance (%) between HPCs and SW while HPCvar shows the HPCs variation coefficient. 75

List of Algorithms

3.1 Enqueue Operation (Single Writer) 42
3.2 Dequeue Operation (Multiple Readers) 43

List of Listings

1 Local NMI Handlers Chain Iteration. 19
2 Install a IDT custom entry 23
3 A simple IRQ stub 25
4 IRQ entry stubs 26
5 Register a fast IRQ API 27
6 Basic fast-IRQ dispatcher 28
7 Example of perf event creation, release and query functions. 31
8 Context Switch Callback based on Tracepoints. 34
9 Context Switch Callback based on Kprobes 37
10 Intercepting Thread Creation/Termination Events. 38
11 Routine to get cpu usage 94

CHAPTER 1
Introduction

Modern computing systems are complex. Historically, the evolution of computing architectures has followed Moore's law principles [109]: provide each new product generation with more transistors and faster working frequencies. However, in the last two decades, the technology bumped into physical limitations [144, 68, 105] pushing the industry to adjust its strategies to meet the hardware capabilities expected by the market. As a result, hardware solutions nowadays are based on sophisticated multi-core designs and deeply structured memory layouts. Furthermore, the ubiquitous integration of specialized components inside processing units makes the concept of heterogeneous systems more pervasive, even in commodity hardware.

At the same time, hardware capabilities aim to support software needs, which, in turn, are driven by the demands of trending topics such as scientific computing, e-commerce, computer vision, artificial intelligence, big data computing and computer graphics. Simultaneously, applications evolve to comply with the new facilities that the industry supply within new hardware generations, giving rise to new programming environments and frameworks meant to achieve the best HW-SW cohesion and reduce the overall complexity of programming. Nevertheless, the two domains are not always aligned. The increasing complexity of computer architecture makes it challenging to provide a transparent interface, often forwarding the burden of dealing with the low-level details directly to software designers.

In such a complex scenario, understanding the actual impact of the code structure and of its execution on the hardware represents a fundamental yet challenging task to carry out crucial operations such as security enforcement, energy optimization and software-performance consolidation.

Actually, the importance of software analysis has brought many vendors to improve the hardware level capabilities provided on board of off-the-shelf processors. More specifically, for several years, we have witnessed the growing diffusion of Performance Monitoring Units (PMUs) as built-in support able to provide valuable hints about micro-architectural activity—hardly perceptible to software approaches—throughout the execution flow of programs. Furthermore, these elements enable a higher level of transparency

to software, also incurring a restrain overhead and external perturbation, if compared with pure software-based approaches.

While PMUs have been available since very old architectures like Intel Pentium and AMD Athlon, their importance has made them see a considerable expansion over the last decades. As a result, nowadays, many modern chipsets leverage these facilities to boost the flexibility of their functionalities. Indeed, supplying a larger set of features makes a tool more versatile, empowering it to deal with different conditions.

A widespread approach concentrates the hardware-based profiling activity on a specific application of interest to analyze its characteristics. Therefore, hardware-level information is generally collected by executing the code within a sterilized environment where minimal external perturbation improves the quality of the analysis. On the one hand, this strategy is advantageous when carrying out program-oriented analysis in order to identify performance issues and culprit code sequences quickly. On the other hand, when an application is actually executed on a hardware platform, it is not isolated, since the coexistence with other applications leads to contention on the hardware, whose final behavior can be no longer correctly represented by the outcomes of per-application analysis. The actual trajectory of the hardware footprint in such a scenario can therefore lead the hardware to be not optimally or unsecurely exploited.

In this respect, the ever-increasing complexity of the ecosystem of software modules and the sliding towards an ever-growing usage of resource sharing (e.g., Cloud application hosting) demands more specific instruments for understanding the actual impact of software on the hardware footprint. Furthermore, a pervasive requirement is to deal with churn. The number of applications in a generic system is highly variable, making it quite difficult to foresee the real workload in a long-time period. As a solution, more recent research lines have highlighted the opportunity to adopt *near real-time* monitoring/tracing activities combined with *in-place online* data exploitation to overcome the limitation of local/offline strategies. Still, it presents new challenges. Even though the well-established profilers offer a rich feature set, their general structure doesn't make them proper data providers for online data consumption and exploitation. They are generally devised to gather application information for *post-mortem* processing which doesn't demand runtime exploitation.

Despite the inherent complexity of acquiring and handling runtime information to optimize applications or system activities, this is something that most software components do still need nowadays. Indeed, performance optimization and security defense are aspects that modern IT systems cannot ignore. Therefore, built-in hardware supports are a key element in providing on-the-fly system-level data while containing the cost of performing

such a data-gather operation. This characteristic plays an important role in online activities such as self-tuning. Notwithstanding, current tools aim for different objectives and limit PMUs assistance without fully exploiting their potential.

In this scenario, this thesis has a twofold intent. On the one side, we want to explore techniques to provide computing systems with adaptive online capabilities based on PMU-provided data. The operating system, and more specifically, its main kernel components, represent the ideal candidate to host this support. Besides indiscriminate access to the underlying hardware features, the low-latency communication among different OS parts is intrinsically advantageous when targeting system-wide explotation of hardware-level footprints. On the other side, we deem that PMUs' characteristics can be exploited beyond the mere profiling activity. Consequently, we investigate methodologies and approaches to employ such support as an on-demand information provider for online data consumption and system level exploitation—such as system reconfiguration.

As we shall describe PMUs are strictly glued to the micro-architectural design introducing an emphasized heterogeneity in both the interface and capabilities each processor model delivers. Therefore, without loss of generality, the contributions we present in this work can be deployed on a system equipped with different processor models. Nevertheless, to decrease the effort required to meticulously combine higher-level software with lower-level capabilities and get the advantage of one of the most advanced implementations available on the market, we built and tested our solution on Intel-based machines. As a final note, a significant effort of this thesis is put into extending the devised solutions to very different application domains, like security, performance and functional tasks. Although such a choice entails a detailed study of the target problem domains, it backs the validity of the PMUs versatility.

The rest of this thesis is structured as follows. Chapter 2 provides an overview of PMU solutions, encompassing their basic and advanced implementations. Furthermore, it examines connected limitations that hinder the adoption of this support as well as intrinsic factors that should be considered when relying on such elements. Also, in the final section of the chapter, we present the baseline state-of-the-art in the area of PMU exploitation. However, given the relatively ample set of applications of PMUs we propose in this thesis, we will provide further discussions on the literature in the chapters specifically focused to the different applications.

Chapter 3 discusses various practical strategies to exploit PMUs beyond post-mortem profiling, suitable for different application contexts. This outcome directly stemmed from this long-journey experience and embraced the internals of the software solutions at the base of the contributions pre-

sented in this thesis. Although all of them deal with different contexts, the adaptability of our solutions goes beyond the proposed application domains. Generality represents one of the main characteristics of our tools, whose scaffold is devised as the combination of essential functional blocks. Furthermore, in the subsequent chapters, we present how to take advantage of the presented strategies to address different challenges.

Chapter 4 presents a kernel-level infrastructure that allows system-wide detection of malicious applications attempting to exploit cache-based side-channel attacks to break the process confinement enforced by standard operating systems. This infrastructure relies on hardware performance counters to collect information at runtime from all applications running on the machine. Furthermore, we derive high-level detection metrics from these measurements to maximize the likelihood of promptly detecting a malicious application.

Chapter 5 introduces a methodology to exploit the hardware-level footprint to assist operating system CPU-scheduling decisions—hence we consolidate CPU-scheduling activities. This support dynamically performs the score evaluation of different co-scheduling strategies by combining micro-architectural information with energy consumption indicators. Consequently, process placement on CPU doesn't follow the statically defined heuristics of the OS, rather it dynamically self-tunes according to the features of the active workload and the processor capabilities.

Chapter 6 promotes a new way of adopting built-in hardware supports to extend the processor computing capabilities. In particular, we present a hardware-assisted memory tracing support that transparently assists roll-back operations in speculative computing—specifically in speculative parallel discrete event simulation.

Chapter 7 concludes this thesis and outlines possible future research work.

The work presented in this thesis is based on the following original contributions:

- **Stefano Carnà**, Serena Ferracci, Emanuele De Santis, Alessandro Pellegrini and Francesco Quaglia, *Hardware-assisted Incremental Checkpointing in Speculative Parallel Discrete Event Simulation.* In Proc. of the 2019 Winter Simulation Conference (WSC'19) - 2019

- **Stefano Carnà**, Serena Ferracci, Alessandro Pellegrini and Francesco Quaglia, *Don't be paranoid: dynamic detection and mitigation for threats exploiting cache-based side-channel attacks.* Advanced Computer Architecture and Compilation for High-performance Embedded Systems (ACACES'20) - 2020

- **Stefano Carnà**, Serena Ferracci, Alessandro Pellegrini and Francesco Quaglia, *Fight Hardware with Hardware: System-wide Detection and Mitigation of Side-Channel Attacks using Performance Counters*. ACM Digital Threats: Research and Practice - 2022. *Accepted for publication*
- **Stefano Carnà**, Alessandro Pellegrini and Francesco Quaglia, *Exploiting Hardware Performance Counters Beyond Profiling*. Software: Practice and Experience. *Under review*
- **Stefano Carnà**, Alessandro Pellegrini and Francesco Quaglia, *CPU-scheduling Consolidation: Application Co-scheduling using Performance Counters. In preparation for journal submission*

Additionally, during my PhD, I contributed to the reproducibility of computational results initiative producing the following publication:

- **Stefano Carnà**, *Reproducibility Report for the Paper: Partial Evaluation via Code Generation for Static Stochastic Reaction Network Models*. In Proc. of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2020

CHAPTER 2

Performance Monitor Units Background

Most modern CPUs are equipped with hardware-based profiling capabilities granted by specialized Performance Monitoring Units. PMU is an Intel term initially used to indicate these hardware-assisted capabilities, but it was later considered an umbrella term to denote any dedicated circuitry implementing monitoring objects. Hardware Performance Counters (HPCs) represent the most common implementation of such units. HPCs were already available on old CPU architectures, such as AMD's Athlon [5], Motorola's PowerPC [45], Compaq's Alpha [32], Sun's UltraSparc I [107], Intel's Pentium [67], and Itanium [139]. They provide a valuable instrument for hardware introspection as a resource for debugging and optimization activities. Unlike other techniques relying on a *time-based* mechanism, which collects processor-state snapshots at regular time intervals, HPCs enable an *event-based sampling* (EBS), allowing a profile generation complying with specific event occurrences.

The events driving the execution of HPC operations are so-called *hardware events*. They define the classes of phenomena in architectural elements by observing the processor state evolution during code execution. Examples of hardware events are the occurrence of a write operation in memory, an L3 cache miss, an L1 line replacement, or the fact that a conditional branch in the execution flow of the program has been taken.

The events that a processor is aware of (and can be triggered) are highly coupled with the actual processor architectural family. This is because the generation of a hardware event is physically triggered by data paths or control signals implemented in the actual control unit of the CPU, and its management is handled by the processor's firmware, which is often subject to partial or complete re-implementation across different families of processing units. In the literature on hardware profiling, this extremely low-level information is often referred to as *micro-architectural events*. The benefit of relying on micro-architectural events is that they can be highly optimized and work at the speed of the CPU. At the same time, micro-architectural

events are directly linked to design choices. Indeed, they may vary among different manufacturer products, accounting for innovative features shipped with new design generations. Also, there is no guarantee that a micro-architectural event can be monitored in two different processor models or can be associated with the same hardware phenomenon.

The complexity of these low-level micro-architectural events is sometimes masked by hardware manufacturers, which try to select a set of events—called *architectural events*—that are deemed common to different architectural models. Relying on architectural events increases the portability of solutions based on this support.

Modern CPUs support hundreds of micro-architectural events, but only a handful of HPCs are actually available[1]. Among the rich set of events that can be monitored, it is possible to find:

1. *Time events*: these are events that can track time evolution. They are often based on a particular hardware counter, often called TimeStamp Counter (TSC)[2], expected to be incremented at each clock cycle. Nevertheless, in many architectures, its activity depends on the processor state, and, in some cases, it may skip some counting.
2. *Instructions progress*: a dedicated counter tracks the retired instructions during CPU activity, providing a primary form of processor throughput—for instructions composed of multiple micro-operations, the counter is incremented when the last micro-operation is retired[3].
3. *Memory access patterns*: caches can be monitored at several levels, and counters can be incremented for each miss/hit event.
4. *Branches*: a counter can provide further information on branch instructions such that it is possible to notice events, such as branch mispredictions or retired branch instructions.

Each HPC can track exactly one micro-architectural event at a time. Despite all the implementation differences, the software interface to control an HPC is typically based on a couple of *model-specific registers* (MSRs):

1. a *selector* (or *control*) register, which is used to specify the HPC operating mode, and the architectural event to be observed;
2. a *counter* register, which is incremented every time the associated architectural event is triggered—of course, the counter can overflow.

The common way to access these counters is through a read/write on model-specific registers (MSRs) exposed by the underlying architecture.

[1] At the time of this writing, almost all off-the-shelf x86 CPUs offer up to 16 general HPCs per physical core.

[2] On x86 architectures it can be accesed by the RDTSC instruction.

[3] On x86, the REP prefix does not effect the number of times the counter is advanced.

MSRs differ from traditional registers (like general-purpose ones) because they are used to configure and toggle specific features on the CPU that may not be present in other models. On x86 architectures, it is possible to operate on MSRs via `rdmsr` and `wrmsr` [67, 5] instructions (which are privileged instructions), or by directly reading the counter from userspace via the `rdpmc` instruction [67].

The control register can be used to activate two different operating modes. When the HPC is configured to work in *counting mode*, the counter register simply accumulates the number of target architectural events observed while any program is running on the CPU. On the other hand, when the HPC is working in *sampling mode*, the system generates a hardware interrupt when the counter register overflows. The operating system handler associated with this interrupt can be programmed to take a CPU snapshot to understand what is currently going on in the profiled application and investigate the context that generated the hardware event.

Nevertheless, when the interrupt service routine is activated, the program counter stored in the interrupt frame on the stack might not be associated with the actual instruction whose execution triggered the overflow of the counter, possibly making the whole approach useless if exact information is desired for profiling. This problem is exacerbated mainly in modern superscalar architectures that rely on out-of-order execution engines to speed up the execution of applications. Here, multiple operations are executed concurrently—as an example, the AMD Fam 10h processor can have up to 72 in-flight operations at any time. Therefore, due to the high number of concurrent pipelines and the different order associated with micro-operation execution and instruction retirement, the actual sampling notification may be late with respect to its generation point. This delay is called *skid*. It essentially states the maximum error interval between the current instruction pointer and the one causing the interrupt firing. As a result, the triggered event may not be precisely associated with the instruction that generated it but with one of its neighbours. Overcoming skid-related side-effects is one of the motivations that led hardware designers to improve the Performance Monitor Units by extending the HPCs with extra advanced capabilities at the expense of increased hardware complexity.

2.1 Intel's Precise Event-Based Sampling

Precise Event-Based Sampling (PEBS) [67, 7] is an Intel extension to traditional HPCs to increase their precision. It is an EBS solution that introduces the concept of *precise events*, i.e., events exposed at a minimal (mostly zero) skid effect. Even though PEBS is built on top HPCs, it does not take complete control over them: it is, therefore, possible to exploit both PEBS and

traditional HPCs simultaneously[4].

PEBS introduces a new hardware-based mechanism that automatically saves the processor context when an active HPC overflows. This solution, called *PEBS assist*, is implemented at the firmware level, and it avoids any code interruption to gather extra processor information related to the event itself—no hardware interrupt is required to save the CPU context, which can be inspected at a later time. PEBS is still based on the usage of the standard counters to work in a combined manner. In particular, a HPC can still be configured to work in sampling mode. In this way, when the counter reaches the configured threshold, a hardware interrupt is fired.

When gathering the event-related data, the information is packed into a structure called *PEBS record* which represents the base element of the *PEBS buffer*. This buffer is located in the *Debug store* (DS) save area, whose size can be defined at setup time by writing into the DS model-specific register. Every time a monitored architectural event triggers the PEBS support, a data sample is produced to snapshot the whole CPU state. The *PEBS index* identifies the next record to be filled with a newly-generated sample in the PEBS buffer. When the PEBS index reaches the *PEBS threshold*, which can be configured at setup time, a hardware interrupt informs the operating system that the buffer is almost full, and a read operation should be carried out as soon as possible to avoid losing samples.

2.2 AMD's Instruction-Based Sampling

Instruction-Based Sampling (IBS) [5, 75] is a precise support that adopts a different methodology for generating data, that in some ways, can be considered as a variation of Time-Based Sampling (TBS). The yardstick of this sampling technique is the instruction[5]: the counter increments every time an instruction is executed and, similarly to the HPC's strategy, an interrupt is sent to the processor as soon as the threshold is reached. The hardware-firmware cooperation is in charge of collecting the CPU state combined with additional information at the precise moment the culprit instruction is fetched from the CPU frontend[6] and following its walk through the entire pipeline until retirement. All the events generated by the instruction execution are recorded into dedicated MSRs, realizing a complete snapshot of all the observed events. IBS defines an orthogonal solution to PEBS, providing

[4]The compatibility among PEBS and available hardware events is an ongoing process that only in the latest versions of the support is not limited to specific domains.

[5]Actually the user can define if the counting process has to be based on either executed micro-operations or elapsed clock cycles.

[6]IBS is split into the *fetch* and the *execution* units that provide information concerning the processor frontend and backend, respectively.

a detailed information that aims for a comprehensive and complete analysis rather than focusing on studying more specific dynamics.

AMD's Lightweight Profiling

Lightweight Profiling (LWP) [5, 3] has been a promising advanced support targeting improvement of the IBS capabilities and introducing several features, some of which were already implemented in competitor's solutions (e.g. buffering in PEBS). Contrarily to performance counters and IBS, LWP focuses only on user activity[7], reducing the overall profiling overhead and enhancing the accuracy of collected data. Compared to PEBS's strategy, it extends the concept of thread context to the data collection, relieving the software of the burden of performing this operation. Consequently, not only different threads can be simultaneously and transparently monitored without conflicts, but even profiled and not profiled processes can run simultaneously without an additional overhead to turn off the support for the second ones. Unfortunately, LWP did not have much success, and even AMD's official support was limited. The presence of some hardware issues (see Section 2.3) and the low users' interest have likely been some of the reasons which led AMD to remove this support from future processor generations—LWP is not available anymore since the first ZEN architecture release—to take advantage of die space to foster other features.

2.3 Technical/Scientific Problems and Needs

Monitoring tools are very old elements that are nowadays employed to accomplish disparate purposes. There are many solutions available in both commercial and research contexts, giving different profiling capabilities and built-in analysis logic. However, if we look for a complete tool to perform prompt actions at runtime, while the process being monitored is still alive, our search won't find an exhaustive result.

The tools combining profiling support with data processing, like for example *perf_events* [100], primarily focus on the latter phase, generally targeting local and static optimization and ignoring the system resource consumption. Conversely, research instruments operating for online system tuning are often designed to accomplish specific tasks, so they cannot be immediately generalized to face other applications. In the following we summarize the problems embedded in currently available solutions, methodologies, and techniques:

[7]LWP works only for code running at Code Protection Level (CPL) 3.

Heisen-Monitoring

The tight correlation between accuracy and cost of profiling may cause the rising of the Heisen-monitoring effect [113]. Such an effect, also known as *Heisenberg Monitoring Uncertainty Principle*, reflects the fact that the more closely you observe something, the more you affect it. Whenever the system is asked for its state, regardless of the analysis objective, the system must suspend the current task to provide the requested context-information snapshot. Of course, data consistency must be kept and the system will resume paused work only when the snapshot is correctly created which, depending on the requested data size, is going to take time. Additionally, if the data is demanded at high frequency, the activity may cause a system overload. This problem not only lowers the performance of the whole execution responsiveness but also impacts the quality of the monitoring accuracy by sullying the data quality with values deriving from monitoring activity itself.

Lack of standard

Even though two architectures designed to provide the same ISA guarantee the same interface between hardware and software, the built-in monitoring support, in each of its variants, may supply different capabilities. These elements work at a lower level and are strictly connected to the design internals which, indeed, may provide different characteristics depending on the considered project. Setting up and managing the support follow model-dependent rules which, commonly, require accessing of a set of *model-specific registers* (MSRs)[8], also demanding detailed knowledge of the hardware at issue. Furthermore, the deeply-rooted nature of these components cannot always guarantee uniform capabilities among different architectural implementations ascribing a low portability level to solutions employing them.

Hardware support flaws and issues

The high complexity of modern computing architectures makes the evolution of processor designs harder and harder, requiring a set of not trivial steps to guarantee several properties: correctness, security, reliability, performance, backward compatibility. Furthermore, some of them turn out to interfere with each other (e.g. security and performance, see Chapter 4) so that introducing a new element or improve an existent one may break the desired behavior of other components. Hardware vendors are aware of the problems that may arise in a new product, some due to a not fully verified

[8]MSRs are special registers used to query and manage peculiar processor features which may be proper to specific architecture design.

design[9], some others stemming from side-effects after physical implementation. Such a drawback reflects on PMUs too. On the one hand, the support may not be able to properly collect profiling information so that the generated data does not fully represent the machine activity. On the other hand, the interaction with the support may not follow the specification rules, requiring ad-hoc procedures to avoid the onset of runtime problems. Even though there is a tight collaboration among vendors and users to find and report these bugs [70, 72, 4, 6, 11, 12], sometimes the workaround may not be implemented through a firmware update but would require a hardware redesign, which is impractical for already produced processors. As a result, in affected models, monitoring supports linger in this unstable state compelling software developers to provide software patches to bypass the hardware flaw, of course reducing the overall effectiveness of the dedicated circuitry. Contributions presented in this thesis consider this problem, and we discuss, at the occurrence, how we tackled it.

Architectural internals knowledge

Although both software- and hardware-based profilers provide proper tools to carry out detailed analysis on different targets, ranging from a wide study of the system activity to a fine-grained inspection of the process execution, the key requirement is the identification of the right metrics, i.e. the quantitative measurements describing the properties of the task to be accomplished. Generally, users need metrics very close to the high-level description of the problem to maximize the effectiveness of the analysis but, especially when dealing with hardware supports, available metrics are not representative information for the selected problem also demanding a high user familiarity degree with the underlying architecture. This question makes the entire process more complicated, resulting most of the time in a considerable effort to identify the right way to design the study.

2.4 Literature Review

Since their initial implementations, Performance Monitor Units have been largely investigated in literature providing a complete spectrum of their capabilities. Even though PMUs can monitor hardware events with high precision, their accuracy is subject to non-determinism effects which highly depend on the combination of the architecture design and the selected metric [85] [165]. Weaver et al. showed in [156] that part of this inaccuracy degree is generated by uncontrollable system phenomenons such as interrupts

[9] Inside a high non-deterministic system, edge cases may be hard to be verified)

or speculative executions[10], thus possibly affecting the profile differently at each run. At the same time, despite their implementation is based on dedicated circuitry, the overhead introduced is not always negligible [115] [155] and can sensibly vary according to the profiling mode, the frequency, and the techniques adopted to collect and read the gathered data. This reinforces the well-known accuracy-overhead relation which at high rates not only slows down the entire system execution making the monitoring activity too invasive for continuous sessions but also introduces side-effects such as profile data perturbation which turn out to tamper with the goodness of the measurement [111] [7].

As described in Section 2.3, PMUs lack of a standard interface and the interaction policies not only differ across processor vendors but can require different actions even among models of the same vendor. As a consequence, most hardware-enhanced software-analysis programs rely on external libraries or tools to nimbly combine functionality and portability of the PMU-based support. OProfile [90], Perfctr [126], Perfmon2 [48], PAPI [23] represents the most used interfaces hooked by software applications to take advantage of hardware supports. PAPI is undoubtedly the most adopted solution whose design evolution is directly supported by major semiconductor companies such as Intel, AMD, ARM, IBM, and Nvidia. However, it provides just a help to merely work with performance counters on several architectures, but the knowledge on how to visualize and interpret generated information is still demanded to the user. AMD code analyst [47], Intel VTune [98], HPCTool [2] and LikWid [146] are advanced tools which do not export a PMU interface, but grant a complete set of functionalities. In fact, they not only rely on a "helpful" graphic UI that drives non-experts during session configuration, but also have a built-in analysis tool that pinpoints critical problems of analyzed execution to carry out disparate tasks such as debugging a performance optimization. Nonetheless, they do not provide any means for external software interaction, limiting their activities to local, though very precise, application profile which cannot be exploited for online tuning. The most versatile framework in the Linux system is perf_events. It is directly integrated inside kernel modules and can be engaged in either system-component activities or user applications to leverage both software and hardware instrumentation. Moreover, it is coupled with a user-space utility accounting for the same high-level capabilities of the aforementioned advanced tools. Even though perf_events has all the makings to support the realization of self-tuning software elements, Weaver [155] demonstrated how its evolution path introduced system performance degradation in favor of more advanced features integration. Generally, profilers do not care

[10]The execution of speculative paths is recorded even if the computing progress is finally dropped, e.g. due to misprediction.

about this drawback because they just gather data in order to perform post-mortem analysis. Even so, this represents a real limitation for techniques acting in an on-line fashion, which requires the identification of the right trade-off between the generated data and the polling frequency of such information, without overloading the system and perturbing original program flow with the profiling activity.

Despite the highlighted problems, hardware monitoring supports have been used to address disparate research challenges. HPCs have been employed to improve *energy efficiency* [106] of systems. In this context, Singh et al. [140] proposed a new methodology to estimated real-time power consumption and devise a power-aware system scheduler.

Wang and Boyle showed that hardware instrumentation can drive compilers to achieve optimal mapping of program parallelism on the underlying architecture by performing automatic profiling runs [154]. Many supports have been designed to diagnose several elusive problems [18] and anomaly detection [13] in a large-scale production system.

Memory management is fundamental to achieve the desired system responsiveness and novel strategies for *runtime memory allocation* on NUMA systems [97] adopted PMUs to retrieve low-overhead and accurate information on allocator choices and memory access paths [87]. However, the solutions are based on either a hybrid approach — which collects data in preliminary analysis and uses results as a hint to conduct the runtime optimization in a next phase [101] — or a pure on-line logic exploiting experimental measurement for parameters tuning which indeed doesn't represent a flexible solution over general application domains [39]. Molka et al. studied the memory footprint of an application detecting its memory-boundedness [108], also highlighting the complexity to choose the right metrics [156]. Hardware introspection provided by this built-in profiling probes have been shown to be suitable to address several security challenges [37][62], ranging from identification of Return-Oriented Programming (ROP) attacks [168] to the detection of several exploits of side-channel vulnerabilities [114] [56]. Furthermore, the design of countermeasures for new *transient execution vulnerabilities* [28] present in most processors required new strategies featured by hardware instrumentation [127] to trickily observe the stealth activities performed by attackers.

CHAPTER 3

Exploiting hardware beyond offline analysis

The evolution of HPCs has always been related to performance profiling, and well-established profilers have extensively used them (e.g., Linux `perf` or Intel V-Tune). In this context, the data collected from HPCs have been chiefly used *post-mortem*: at the end of the profiled application's execution, performance values are collected from HPCs, and later analyzed to construct a program's profile. In the case of more advanced flavours of HPCs, such as the Intel PEBS mentioned above, this post-mortem activity might also entail scanning through a memory area that keeps the more detailed samples generated during the execution. This strict separation between sample generation and sample collection/analysis is one of the fundamental reasons why HPCs are considered a lightweight solution to profiling. Indeed, as mentioned before, the generation of samples relies on dedicated data paths and firmware on the CPUs, making it efficient. Conversely, sample acquisition and analysis (which can be regarded as the most time-consuming step) is made *after* the application's execution is completed. This two-phase approach has the additional benefit of inducing minimal interference to the application's execution, thus avoiding the so-called *Heisen-monitoring* effect.

Nevertheless, more recent research lines have highlighted the opportunity to rely on HPCs for *online* self-tuning of applications or to carry out *near real-time* monitoring activities. In this scenario, the capability to effectively monitor the behaviour of applications using non-intrusive facilities enables new applications of the Autonomic Computing paradigm [81, 60]. In this direction, HPCs have been used in a myriad of applications, such as dynamic software profiling [10], children privacy protection [16], failure prediction in computing systems [117], random number generators [143], or for the generation of test programs via fuzzing to enhance code coverage [37] just to mention a few diverse applications.

These solutions share the need for an efficient analysis of the generated performance samples, which is the activity associated with the most con-

siderable overhead in traditional profiling applications. Indeed, while HPCs are fast and non-intrusive at generating samples, consuming these samples to create higher-level information to be exploited by any online application is costly.

The overhead introduced in the system to collect HPC samples is thus relevant for many modern applications. This overhead is exacerbated if an application requires to collect these samples at a higher frequency. Therefore, great care must be taken to control the collection frequency (otherwise, thrashing phenomena might be experienced), or it becomes fundamental to devise less-intrusive solutions to gather the samples.

At the same time, there is still the need to face other problems such as those described in Chapter 2. Althought, many of these aspects are explicitly dealt with by standard tools exploiting HPCs, such as Linux `perf` or Intel V-Tune, yet, if an application wants to perform online optimisations based on data gathered from HPCs, relying on these post-mortem profiling frameworks might incur too high overhead.

In this chapter, we discuss several approaches which can be used to program, control, and read data from HPCs. Our strategies try to solve some or all of the limitations mentioned above or risks related to the usage of HPCs, thus serving as building blocks for online monitoring/self-tuning systems. Every approach is accompanied by code samples, enabling developers and practitioners to find a comprehensive guide to HPC usage in the context of online exploitation of the samples, e.g., for self-optimization. We couple our reference implementations with an extensive performance assessment, which shows the benefits and limitations of each solution. We tailor our solutions for x86 architectures, encompassing both Intel and AMD chipsets. Moreover, our approaches can be easily ported to other architectures, although various CPU models present technical differences.

3.1 Efficient Collection of PMU Data

As we already introduced, one of the main problems affecting profiling accuracy is collecting the profile data themselves. Typically, profiling agents share the computational resources with the monitored application, and perfectly separating the execution effects can be arduous. Thus, the ability to reduce data pollution, minimising both the *direct* and *indirect* impact of the profiling activity, represents one of the critical capabilities demanded from a profiler. On the one hand, the direct influence is more straightforward to be recognised and, consequently, less complicated to be mitigated. It refers to the observation of the profiling agent code execution and its recording within the under-construction profile combined with targeted application data. Indeed, this pitfall introduces an additional variance (combined with

that coming from other sources), propagating its effects to the following steps, such as data analysis.

On the other hand, correctly identifying indirect perturbations can be convoluted because they are related to actions that produce side effects on the hardware state, altering the system conditions for the other processes. This phenomenon is intrinsic to almost all high-end processor architectures whose CPU time sharing represents a practical example. Although it can transparently guarantee isolation and resource assignment to all the processes from a software perspective, this comes at a cost at the hardware level. Cache pollution is part of it. The cache state may significantly vary between the moment at which a process is preëmpted and when it is rescheduled on the same core, introducing locality penalties to reload data replaced during another process's execution.

In this context, it is essential mastering the mechanisms that regulate the software/hardware interaction to fully exploit the PMU support while minimising the aforementioned undesired effects. Software routines can access PMU registers only running at the highest privilege, namely ring 0 on x86 processors. This constraint holds for almost every support except the HPC counter register that can be read from userspace under certain conditions via the specialised `rdpmc` instruction—we discuss in Section 3.3.4 how to interact with HPCs from userspace.

We can consider running at ring 0 a fundamental requisite to unleash the full PMU potential. When running in kernel mode, there are two main ways to access data collected by these units: polling and interrupting. As the name suggests, the polling mode tries to access the support at some defined points in time, trying to read the information that may not be ready or too rich to discriminate factors generating a particular event. Furthermore, polling reads can happen only during kernel-mode execution, whose transition stems from an interrupt generation or a system call invocation. All in all, we can consider the polling mode as a subset of the interrupt-based strategy with minor frequency control. The main advantage is that it does not (directly) pay the cost of a ring change because it exploits the ongoing kernel-mode execution.

Nevertheless, the generation of an interrupt, namely a *performance monitor interrupt* (PMI), is typically the most used strategy. It guarantees better control of the PMU activity and related overhead. Moreover, the period between two PMIs can be easily configured by setting up the dedicated registers adequately at a single interrupt basis.

Generally, PMUs are configured to generate PMIs on the Non-Maskable Interrupt (NMI) line. This vector is commonly used to register routines linked to exceptional hardware events related to low-level machine management or more complicated hardware faults. Consequently, it demands

a more advanced software routine than an ordinary interrupt line to handle different events. On the x86 architecture, each core has a single non-maskable interrupt line[1]. Linux implements the NMI handler as a dispatcher that iterates over a chain of handlers whose elements are explicitly registered. The NMI routine starts walking through the chain upon an NMI generation, and all the enrolled handlers get called until one handles the interrupt. Each one shall assess the interrupt causes by querying specific registers (or memory locations), revealing whether the NMI was intended for it or another handler. For instance, the PMI handler must check fixed bits in specific MSRs to determine if HPCs or other supports overflowed and thus generated the NMI.

Of course, NMI management can be complicated [136], and some conditions may arise, causing concurrent execution of NMI handlers. Nonetheless, the largest part of the management of these issues is done by the Linux kernel general NMI management subsystem, while the only responsibility left to the interrupt handler programmer is to implement the associated logic appropriately. On the one hand, NMIs provide excellent flexibility because, in the general case, they can preëmpt other interrupts, ensuring timely processing of the profiling data coming from PMUs. In contrast, a priority-based policy regulates the normal interrupts precedence. On the other hand, the execution of the NMI handler requires the development of more code and introduces an overhead that may cause a delay in all planned activities, especially at high PMI frequencies. Furthermore, the NMI generation should be deemed as an extraordinary event, and paying this articulate handler's cost may be helpful only when the NMI priority plays a crucial role.

As mentioned, PMUs have been designed to support debugging and punctual performance optimisation sessions in an offline manner. In this context, pinpointing unoptimised code sections or the execution bottlenecks can be perpetrated through different runs by exploiting a top-down approach that accurately marks the boundaries of the culprit instructions at every instance execution. This flow allows the analyser to adjust the profiling parameters, e.g., sampling rate and data amount, by analysing the previous measurement outcome and then adapting the next session to pursue the code of interest. The prerequisites are different when dealing with an online data collection that points to a complex system and feeds routines that act in place while the observed dynamics are ongoing. In this setting, the *system overhead*, the *sampling frequency* and the *data processing* start playing an essential role and demand a combined optimisation because they influence each other.

Directly modifying the interrupt management strategy can be an effec-

[1]All NMIs go into interrupt line 2.

Listing 1 Local NMI Handlers Chain Iteration.

```
static int nmi_handle(unsigned int type, struct pt_regs *regs)
{
        struct nmi_desc *desc = nmi_to_desc(type);
        struct nmiaction *a;
        int handled = 0;

        rcu_read_lock();

        /*
         * NMIs are edge-triggered, which means if you have enough
         * of them concurrently, you can lose some because only one
         * can be latched at any given time.  Walk the whole list
         * to handle those situations.
         */
        list_for_each_entry_rcu(a, &desc->head, list) {
                int thishandled;
                u64 delta;

                delta = sched_clock();
                thishandled = a->handler(type, regs);
                handled += thishandled;
                delta = sched_clock() - delta;
                trace_nmi_handler(a->handler, (int)delta, thishandled);

                nmi_check_duration(a, delta);
        }

        rcu_read_unlock();

        /* return total number of NMI events handled */
        return handled;
}
```

tive intervention to achieve a more reliable system response. In particular, we want to minimise the *interrupt latency* that, in this case, not only denotes the hardware delay to deliver the interrupt request to the operating system but also encompasses the total number of instructions executed both before the actual management routine is activated and at its end, to resume the preëmpted flow.

Listing 1 shows the platform-independent entry point for the management of NMIs in Linux, which is executed after the platform-dependent assembly preamble[2]. By design, multiple NMIs can fire simultaneously, but their edge-triggered nature allows detecting at most two pending NMIs. This limit requires keeping iterating the handlers' chain even if some called routines have already recognised the interrupt as intended, thus performing some action. An optimisation discerns between *local* and *non-CPU-specific* NMIs[3]. The former has a higher priority, while the latter is considered only

[2]The NMI assembly trampoline is more expensive than conventional IRQ ones because it has to manage broader scenarios and requires more checks.

[3]An NMI can refer to both CPU-internal dynamics, thus requiring the direct action of the interested CPU or system events which any CPU may handle.

if no local NMI has been detected.

The kernel's sole clue about possible concurrent NMIs comes from the result returned by each chained handler, which checks its conditions and returns a value representing the number of processed events. With this method, the kernel catches and resolves the extra NMIs besides the two detected ones and configures some status information to manage the secondary NMI handling. However, an NMI is conceived to signal extraordinary events, and we may assume that they are not very frequent. Indeed, boundary cases are critical, and performing a meticulous check can be crucial in a production system. Unfortunately, this has a cost that directly influences the experienced system overhead. NMI handler subscribers may reach a consistent number in a complex environment where several kernel modules and hardware facilities co-exist, increasing the price to process an NMI on a CPU. PMUs naturally fit profiling operations by design, but they are regulated by a working period whose length sets the interrupt frequency. Thus, fine-grain monitoring sessions demand a high PMI generation frequency which, if mapping on the NMI line, gets a disadvantage due to the already described mechanism.

A less widespread solution is redirecting the PMI to a generic IRQ line by adequately configuring the associated APIC entry. The PMI benefits from a dedicated line and can avoid extra work such as chain inspection. Even though an interrupt can arise on that configured line only when a particular PMU's condition is met, it is a standard practice to keep checking the generation condition by querying the relative MSRs. This check is due to an uncertainty level that affects the correct behaviour of this specialised circuitry, such that spurious interrupt may be generated under some circumstances [22]. As a drawback, the PMI handler's promptness may get thrown on the back burner during the execution of a higher-priority interrupt. Still, these events are relatively rare in real scenarios and may matter only for exceptional occurrences.

Compared to the NMI code, the generic IRQ entry routine comprises fewer instructions for the platform-independent handler function. From a high-level perspective, it performs the following operations:

- check if we are coming from a user- or a kernel-mode and, if needed, swap the GS segment;
- create a complete registers snapshot and switch to the dedicated IRQ stack;
- call the `do_IRQ` function, which sets some system flags and execute the actual handler;
- leave the IRQ stack;

- call kernel housekeeping functions according to the execution mode we are switching back;
- restore the registers snapshot and perform `swapgs` if required;
- return from interrupt.

Additional actions may take place, such as an extra trampoline stack switch or security procedures mitigating hardware pitfalls (such as Meltdown discussed in Chapter 4). Those are present in the NMI assembly routine as well.

Hence, by setting a standard IRQ line as a vector for the PMI, the system can benefit from a shorter code to reach the handler function and a dedicated routine that avoids the previously mentioned chain lookup. These are key factors to reduce the interrupt latency and, consequently, the side-effects linked to the execution of the additional instructions. Linux has dedicated APIs to install a custom IRQ handler in the system. However, they are strictly devised for external components, e.g., PCI-bus-connected devices, which exploit a logical and shared interrupt mechanism that virtually enhances the IRQ line over the traditional value of 256 in x86 machines. Moreover, newer kernel versions have a redesigned `do_IRQ` function to achieve better performance, expecting access to symbols not exported in the mainline version of the kernel. This restriction undermines reprogramming the PMI on a different line inside an out-of-tree kernel module, making the hack impractical without directly working on the kernel source code.

Exploiting a Custom IDT Entry Using a Kernel Module

An effective way to bypass the limits dictated by the Linux Kernel APIs is to directly install the PMI handler into the *Interrupt Descriptor Table* (IDT). Nevertheless, switching from an NMI to a regular interrupt for PMU data collection is not a free lunch. Indeed, several technical challenges arise for this purpose. We discuss in the following a practical approach to reading PMU data by relying on a Linux loadable kernel module (LKM).

On x86 architectures, the number of available IDT vectors is limited to 256, each associated with a dedicated management routine. The IDT can be placed anywhere in the linear address space, and its location is stored in the dedicated IDTR register via the `lidt` instruction. It is a privileged instruction that can be executed only at ring 0, while the IDTR register's value can be read using the `sidt` instruction running at any privilege level. Therefore, to install a custom interrupt handler, we must run in kernel mode—hence the need to rely on an LKM. Moreover, on 64-bit systems, gate descriptors stored in the IDT are an extension of the 32-bit ones, as depicted

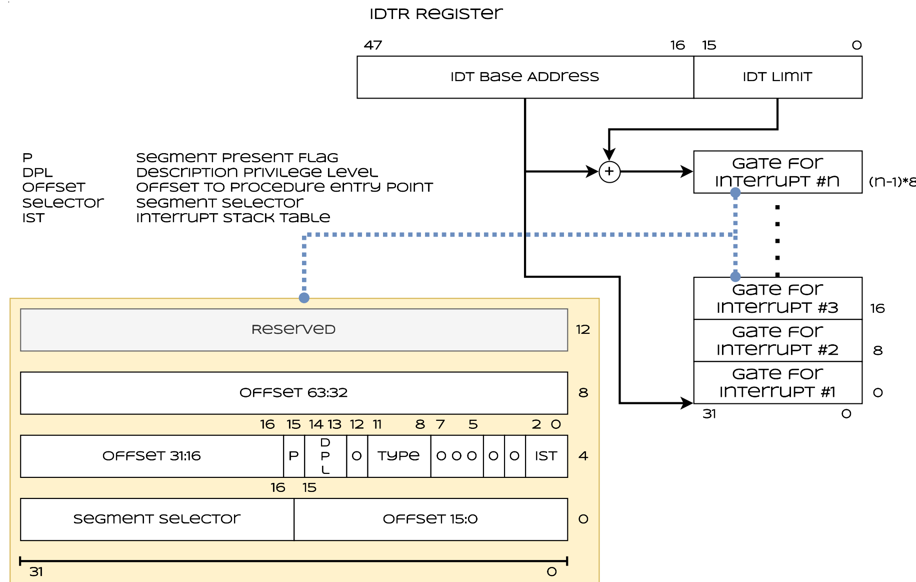


FIGURE 3.1: *The general structure of IDT.*

in Figure 3.1. As a result, the descriptor shows a non-linear structure that might be confusing, in which the address of the associated management routine is not contiguous. This design choice impacts the complexity of the code to manipulate the IDT and the ability to safely modify entries when the kernel has finished loading in memory. The possibly unsafe behaviour when installing an IRQ routine in the IDT at steady state is because 64-bit x86 architectures only ensure writes up to 8 bytes to be atomic. The scattered nature of the interrupt handler's address makes it impossible to rewrite ad IDT line using a single machine instruction. Therefore, concurrent access to that vector may trigger a kernel-mode segmentation fault, causing the system to crash.

Therefore, the challenge is to update the gate where we want to install the custom PMU interrupt handler preventing the CPUs from accessing a partially-modified entry. A straightforward solution can be disabling the interrupt source, namely the PMU's ability to generate a PMI, dodging the concurrency issue. On the one hand, that approach may achieve the desired result in a controlled environment where we are sure that no other profiling agents would interfere with this operation[4]. On the other hand, each CPU core has its own IDTR register on SMP machines. Although this design allows defining a different IDT for each CPU core, Linux has a unique IDT instance installed at system initialisation. Thus, directly modifying the system's shared IDT version would require coordinating all CPU cores to

[4]Of course, this approach is not suitable for a general update of any IDT gate descriptor, especially in the case of uncontrollable events, e.g., page fault exceptions

inhibit the interrupt source until the procedure finalisation, which, on large multicore systems, may produce a non-negligible overhead.

Listing 2 Install a IDT custom entry

```
1  static void smp_load_idt(void *addr)
2  {
3          struct desc_ptr *idtr;
4          idtr = (struct desc_ptr *)addr;
5          load_idt(idtr);
6  }
7
8  static struct desc_ptr *clone_current_idt(void)
9  {
10         struct desc_ptr *idtr;
11         void *idt_table;
12
13         idtr = (struct desc_ptr *)kmalloc(sizeof(struct desc_ptr), GFP_KERNEL);
14         if (!idtr)
15                 return 0;
16
17         idt_table = (void *)__get_free_page(GFP_KERNEL);
18         if (!idt_table)
19                 return 0;
20
21         store_idt(idtr);
22
23         memcpy(idt_table, (void *)(idtr->address), PAGE_SIZE);
24
25         idtr->address = (unsigned long)idt_table;
26         return idtr;
27 }
28
29 void idt_patcher_install_entry(unsigned long handler, unsigned vector,
30                                unsigned dpl)
31 {
32         gate_desc new_gate;
33         gate_desc *new_idt, *cur_idt;
34
35         /** FIX THIS CODE **/
36         cur_idt = (gate_desc *)patched_idt[patched_idt_idx]->address;
37         patched_idt_idx ^= IDX_STEP;
38         new_idt = (gate_desc *)patched_idt[patched_idt_idx]->address;
39
40         // Create the entry in the spare IDT instance
41         pack_gate(&new_gate, GATE_INTERRUPT, handler, dpl, 0, 0);
42         write_idt_entry(new_idt, vector, &new_gate);
43
44         // Set the spare IDT instance as system's one
45         on_each_cpu(smp_load_idt, patched_idt[patched_idt_idx], 1);
46
47         // Upload old IDT instance
48         write_idt_entry(cur_idt, vector, &new_gate);
49 }
```

An alternative approach can be based on identifying an alternative way to perform an atomic update of the IDT data structure. Indeed, the execution of the `lidt` instruction can be regarded as the serialisation point for the update operation. To this end, it is possible to create a full verbatim copy of the currently in-place IDT table and modify an entry to install the PMI

handler while the original table is used to serve interrupt requests. This approach allows us to avoid any concurrency problems. Once the new IDT instance is ready, we can force all CPU cores to reload their IDTR register to point to the new table simultaneously. This synchronous update is required to prevent different interrupt handler versions from concurrently executing on different CPU cores. It can be easily implemented, as shown in Listing 2, by relying on the built-in cross-core messaging mechanism based on *Inter-Processor Interrupts* (IPI). In the provided example, the `on_each_cpu()` function call forces all CPU cores to execute the `smp_load_idt()` function, which the new version of the IDT.

To complete the description of our approach, we have to discuss the interrupt routine to install in the IDT entry to process the PMIs. As described previously, the typical approach in Linux is to have a shared assembly stub that later calls the appropriate platform-independent function to process the IRQ, ensuring a logical separation among architecture-dependent and independent code. Nesting our custom PMI handler in the Linux IRQ dispatcher from a kernel module requires tampering with many kernel layers, making the approach too fragile while also having to account for all the possible code variants across the different kernel versions. Considering that we only target the x86 platform, a simple approach is to bypass the shared stub and make the IDT entry point to a custom platform-dependent dispatcher, which we provide in Listing 3. This simple dispatcher performs standard activities also carried out by the Linux kernel for general IRQ management, although some actions are simplified (e.g., not all general-purpose registers should be saved) due to its special-purpose nature. This IRQ entry point can then call a C function (`custom_pmi_handler` in the example listing) that can read profiling data from the PMUs that caused the PMI firing.

As a last note, versions of the kernel that have Meltdown mitigations active (such as Kernel Page Table Isolation, KPTI [54]) must be handled with additional care. Indeed, the strict kernel and userspace isolation makes it impossible for the hardware to see any content of the loaded LKM when running in userspace. If a PMI is fired when the system is running in user mode, the firmware will not find the PMI handler entry point mapped in memory, causing a fault. Patching the globally-mapped *cpu entry area* used by Linux to give access to fundamental parts of the kernel to the firmware is not an option because it is defined at kernel compile time. Therefore, the only practical solution to make the proposed LKM-based approach work within a PTI-enhanced kernel is to disable KPTI at boot time by setting the `nopti` or `mitigations=off` kernel parameters. If lowering system security cannot be considered an option, a different path should be taken, which we shall discuss in the following section.

Listing 3 A simple IRQ stub

```
1  extern void pmi_irq_entry(void);
2
3  asm(".globl pmi_irq_entry\n"
4      "pmi_irq_entry:\n"
5      "        cld\n"
6      "        testq $3,8(%rsp)\n"
7      "        jz 1f\n"
8      "        swapgs\n"
9      "1:\n"
10     /* Push error code */
11     "        pushq $0\n"
12     /* Push CPU registers */
13     "        pushq %rdi\n"
14     "        pushq %rsi\n"
15     /* ... omit the complete push sequence */
16     "        pushq %r14\n"
17     "        pushq %r15\n"
18     /* Call high-level handler */
19     "        call pmi_irq_handler\n"
20     /* Pop CPU registers */
21     "        popq %r15\n"
22     "        popq %r14\n"
23     /* ... omit the complete pop sequence */
24     "        popq %rsi\n"
25     "        popq %rdi\n"
26     /* Clean error code */
27     "        addq $8,%rsp\n"
28     "        testq $3,8(%rsp)\n"
29     "        jz 2f\n"
30     "        swapgs\n"
31     "2:\n"
32     "        iretq");
```

Kernel-level Fast IRQ Interface

The Linux Kernel has been designed to provide elevated flexibility and modularity to support dynamic extension via the runtime loading of out-of-tree modules. Although this capability allows implementing functions for direct communication with the kernel internals, it is limited to the public interface and exposed symbols that sometimes are not comprehensive for the planned task. Such a design choice is justifiable, as it is part of the hardening process that ensures the stability and reliability of the operating system itself. Indeed, the IDT management and related data structures belong to the *non-exported objects*. However, as discussed in the previous section, it can be desirable to install special-purpose IRQ management routines for PMIs. Here, we discuss a patch to the Linux kernel source that allows exposing to LKMs an interface to install custom IRQ handlers. This interface can be used to pursue a goal similar to the one discussed in Section 3.1 also when security enhancement patches are active.

Our patch is based on the organisation of the `entry_64.S` source file, which contains the most significant part of the platform-dependent code.

We are particularly interested in the interrupt/exception entry stubs for our purposes. We report in Listing 4 the relevant part of the code which is in charge of generating all the 8-byte stubs starting from the `irq_entries_start` symbol. During the final part of the IDT initialisation at kernel startup, the stubs are linearly scanned starting from this symbol, and the entries which are actually installed in the IDT are marked in the `system_vectors` bitmap. Each stub is in charge of pushing the corresponding vector number to the stack and jumping to a different routine (either `common_interrupt` or `common_fast_interrupt`, depending on the nature of the IRQ entry). In all cases, the amount of work carried by the stubs is minimal, meeting our need for a low-overhead activation of the PMI handler.

Listing 4 IRQ entry stubs

```
1          .align 8
2  ENTRY(irq_entries_start)
3      vector=FIRST_EXTERNAL_VECTOR
4      .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
5          UNWIND_HINT_IRET_REGS
6          /* Note: always in signed byte range */
7          pushq       $(~vector+0x80)
8          jmp        common_interrupt
9          .align       8
10         vector=vector+1
11     .endr
12     /* Place the fast irq entries just after the irq entries */
13     /* fast_irq_entries_start */
14     vector=FIRST_EXTERNAL_VECTOR
15     .rept (NR_VECTORS - FIRST_EXTERNAL_VECTOR)
16         UNWIND_HINT_IRET_REGS
17         /* Note: always in signed byte range */
18         pushq       $(~vector+0x80)
19         jmp        common_fast_interrupt
20         .align       8
21         vector=vector+1
22     .endr
23 END(irq_entries_start)
```

Even though it is feasible to manually craft a dedicated IRQ stub and assign it to a free vector during the IDT filling (using an approach similar to the one described in Section 3.1), it would require hardcoding the PMI line or, at least, a mechanism to discover possible incompatibilities with other elements. Instead, we propose an API that can work with any line in the interval `FIRST_EXTERNAL_VECTOR`–`FIRST_SYSTEM_VECTOR`. Our strategy leverages the contiguous placement of the two groups of stubs and the fact that the two loops do not fill the entire IDT table (i.e., by default, less than 256 vectors are used by Linux)

Therefore, we can determine an offset at compile time that allows us to identify a free vector line in the IDT. This line can be used to install a new custom fast IRQ handler. In Listing 5 we show the implementation of

Listing 5 Register a fast IRQ API

```
int request_fast_irq(unsigned int fast_irq, fast_handler_t handler)
{
        unsigned long fast_IRQ_base;
        char *fast_irq_entries_start =
                irq_entries_start +
                8 * (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR);

        if (fast_irq < FIRST_EXTERNAL_VECTOR || fast_irq >= NR_VECTORS) {
                pr_warn("Cannot install irq handler at %u. Invalid\n",
                        fast_irq);
                return -EINVAL;
        }

        if (test_bit(fast_irq, system_vectors) ||
            fast_vector_handlers[fast_irq]) {
                pr_warn("Cannot install irq handler at %u. Busy\n", fast_irq);
                return -EBUSY;
        }

        /* The first vector is placed at index FIRST_EXTERNAL_VECTOR */
        fast_IRQ_base = (unsigned long)(fast_irq_entries_start +
                                        8 * (fast_irq - FIRST_EXTERNAL_VECTOR));

        /* Magic trick, setup the new entry */
        idt_table[fast_irq].offset_low = (u16)fast_IRQ_base;

        set_bit(fast_irq, system_vectors);
        fast_vector_handlers[fast_irq] = (unsigned long)handler;

        return fast_irq;
}
EXPORT_SYMBOL(request_fast_irq);

int free_fast_irq(unsigned int fast_irq)
{
        /* The first vector is placed at index FIRST_EXTERNAL_VECTOR */
        unsigned long old_IRQ_base =
                (unsigned long)(irq_entries_start +
                                8 * (fast_irq - FIRST_EXTERNAL_VECTOR));

        if (fast_irq < FIRST_EXTERNAL_VECTOR || fast_irq >= NR_VECTORS) {
                pr_warn("Cannot uninstall irq handler at %u. Invalid\n",
                        fast_irq);
                return -EINVAL;
        }

        if (!fast_vector_handlers[fast_irq]) {
                pr_warn("Cannot uninstall irq handler at %u. Empty\n",
                        fast_irq);
                return -EINVAL;
        }

        /* Restore the old handler address */
        idt_table[fast_irq].offset_low = (u16)old_IRQ_base;

        clear_bit(fast_irq, system_vectors);
        fast_vector_handlers[fast_irq] = 0UL;
        return 0;
}
EXPORT_SYMBOL(free_fast_irq);
```

our API that serves this purpose. Our code takes a `fast_irq` vector number and a function handler as parameters. It performs some correctness checks and verifies the system state not to overwrite an already online or previously configured vector line. Note that the `fast_irq_entries_start` indicates the address of the first fast-IRQ stub, which is associated with the first unused entry in the IDT. Considering the number and the size of generated stub entries, they can be allocated within a single 4K memory page, thus demanding only 12 bits to be addressed. Furthermore, the memory contiguity ensures that they all share the upper part of their address. Consequently, our API updates only one of the 16 least significant address bits, which can be implemented as a single atomic write. Such an operation shifts all the vector by 2^X pages, where $X = Y - 12$ and Y is the flipped bit within the lowest offset (lines 25-27). Once the IDT vector correctly points to the desired stub, the line is marked as used on a system-wide basis, and the handler function is saved in a unique system vector which is accessed by a custom pre-handler that we have called `do_fast_IRQ` routine. This mechanism emulates the `do_IRQ` strategy that dispatches the platform-independent function according to the vector number provided by the entry assembly code. To uninstall a fast IRQ, we provide the `free_fast_irq` API function. It is in charge of restoring the system state by uninstalling the `fast_irq` vector, which can be achieved by reverting the address expressed by the selected IDT gate descriptor to the original interrupt entry stub, informing the system that the vector is now available, and cleaning up the corresponding fast-IRQ system vector entry.

Listing 6 Basic fast-IRQ dispatcher

```
1   __visible void __irq_entry do_fast_IRQ(struct pt_regs *regs)
2   {
3           u8 vector = ~regs->orig_ax;
4           struct pt_regs *old_regs = set_irq_regs(regs);
5
6           entering_irq();
7           /* Add guards to safely remove an handler */
8           (*(fast_handler_t *)(fast_vector_handlers[vector]))();
9           /* The IRQ handler is in charge of calling APIC EOI */
10          exiting_irq();
11          set_irq_regs(old_regs);
12  }
```

To complete the description of our approach, Listing 6 shows the basic implementation of the `do_fast_IRQ` function. As mentioned, its main purpose is to pick the right handler to manage the interrupt. It retrieves the vector number from the `struct pt_regs` and directly uses the vector number as an offset in the `fast_vector_handlers` table, which holds the installed handlers.

Overall, relying on the Fast IRQ registration API makes it possible to

quickly install an IRQ handling routine that follows the same approach as regular handlers while following an execution path that requires executing a reduced number of instructions. This is because many corner cases related to IRQ management are not considered in our execution path. In this sense, our approach is not meant to replace the standard Linux management of IRQs, but only to provide an interface to be exploited in non-standard contexts, such as the high-frequency sampling with the PMUs. Moreover, our approach makes it possible to install a fast IRQ management routine after system initialisation, thus making it possible for a kernel module to register/unregister a PMI handler any time it is needed.

3.2 Discriminating the Profiling Domain

PMUs can only directly observe the general system-wide behaviour. Therefore, extracting precise profiling information related to a specific application can become cumbersome—PMUs work at the architectural level, thus ignoring dynamics such as context switches. Indeed, in a time-sharing system, discriminating what application was running when a PMU captured a particular event might be even impossible in a post-mortem analysis. The typical approach adopted while capturing profiling data is to minimise the cross-application noise by drastically reducing the number of active processes, possibly also limiting the operating system's activity. If this kind of system's control is feasible, it is possible to build an acceptably accurate profile. Conversely, since we target online analysis on production systems, we can expect this kind of control to be impossible, thus incurring significant noise in the measurements. *Thread discrimination* is anyhow an essential feature for every profiling system, be it online or offline.

Per-thread Profiling

In *sampling mode*, the sampling period regulates the profiling frequency according to one or more system events, such as retired instructions and elapsed clock cycles. Upon counter's overflow, a PMI interrupt forces the CPU to undertake all the actions required to start the sample collection. This procedure implies stopping the current thread's execution, favouring the previously outlined dedicated routines. Suppose a theoretical execution in which no sample has been generated yet, and no context switch occurs. In that case, once the PMI is fired after a certain amount of time, we will only find samples (if any) associated with the current thread. This means that it is possible to implement per-thread filtering of the generated samples provided that: i) we can associate a trace of generated samples with the current thread; ii) we can intercept a context switch to empty the sample

trace.

While point i) above can be simple, in our reference scenario focused on online self-tuning optimisation, it is unlikely that the monitoring agent should profile any active thread. Therefore, we must also introduce a mechanism to let the system know the pool of threads that the agent should actively monitor. We propose an explicit "registering" procedure, which allows the monitoring agent to collect data related to specific threads (based on `pid` numbers in Linux) or to multi-threaded processes (based on `tgids`). For simplicity, in the following, we generically use the term thread for both cases, with no loss of generality.

In order to reduce the monitoring overhead, an important point is related to the number of registered threads compared to the number of active threads. Considering that we target production systems, we can expect the number of monitored threads to be far lower than the running ones. Therefore, the monitoring agent should leverage some efficient data structure to perform thread lookups, mainly since we target high-frequency sample collection. For this purpose, we have explored four different solutions—we provide performance data in Section 3.4.

The first solution is based on a global shared data structure that can be updated whenever some thread is registered for monitoring. From our empirical experience, hash tables well fit the performance requirements under general settings. To determine whether a thread should be profiled, the monitoring agent can check whether its ID is stored in the hash table. Nevertheless, thread safety must be ensured since every CPU has its own PMUs. Given the high dynamicity that we target, a solution based on locks would negatively impact performance, making this approach non-viable.

A second solution exploits the already-available perf_events subsystem in Linux, an advanced analytics suite available in the Linux Kernel. The tight integration with the Linux kernel is also manifested in the presence of dedicated fields within the `struct task_struct` that we can use to flag a thread as under monitoring. We therefore take advantage of the possibility to register a so-called *software dummy event* for each process we want to monitor. This event is not actually associated with any profiling rule but is used only to mark a thread as under profiling. Furthermore, by registering this custom event without associating any profiling rule, we can leverage its instance as a container of custom data—we will discuss in more detail how to store PMU data on a per-process basis in Section 3.3. Listing 7 shows an example of how to register, deregister and query a perf event. Once the perf event is instantiated, it is queued into the dedicated per-process perf_event list. To check whether a process should be monitored, the monitoring agent can search the dummy event in the corresponding list and check the related private data. Even though all the operations are kept consistent through a

mutex, our experience shows that no significant overhead is introduced.

Listing 7 Example of perf event creation, release and query functions.

```
bool create_task_data(struct task_struct *task, void *data)
{
	struct perf_event *event;
	struct perf_event_attr attr = {
		.type = PERF_TYPE_SOFTWARE,
		.config = PERF_COUNT_SW_DUMMY,
		.size = sizeof(attr),
		.disabled = true,
	};

	event = perf_event_create_kernel_counter(&attr, -1, task, NULL, NULL);

	if (IS_ERR(event))
		return false;
	// Insert custom data inside event
	event->pmu_private = data;

	mutex_lock(&task->perf_event_mutex);
	list_add_tail(&event->owner_entry, &task->perf_event_list);
	mutex_unlock(&task->perf_event_mutex);

	return true;
}

void destroy_task_data(struct task_struct *task)
{
	struct perf_event *event;

	event = __get_perf_event(task);
	perf_event_release_kernel(event);

	mutex_lock(&task->perf_event_mutex);
	list_del(&event->owner_entry);
	mutex_unlock(&task->perf_event_mutex);
}

static void *is_profiled(struct task_struct *task)
{
	struct perf_event *event;

	mutex_lock(&task->perf_event_mutex);
	event = list_first_entry_or_null(&(task->perf_event_list),
					struct perf_event, owner_entry);
	mutex_unlock(&task->perf_event_mutex);

	return event ? (void *)event->pmu_private : NULL;
}
```

The third solution is based on storing a monitoring flag directly on the kernel stack of the profiled thread. Figure 3.2 depicts the overall schema under two different circumstances. The general idea is to use the last available stack entry to store the profile status for the corresponding thread. In this case, determining whether a thread should be profiled involves accessing the process control block (PCB) of the currently-scheduled thread via the `current` macro. The PCB contains a reference to the kernel stack

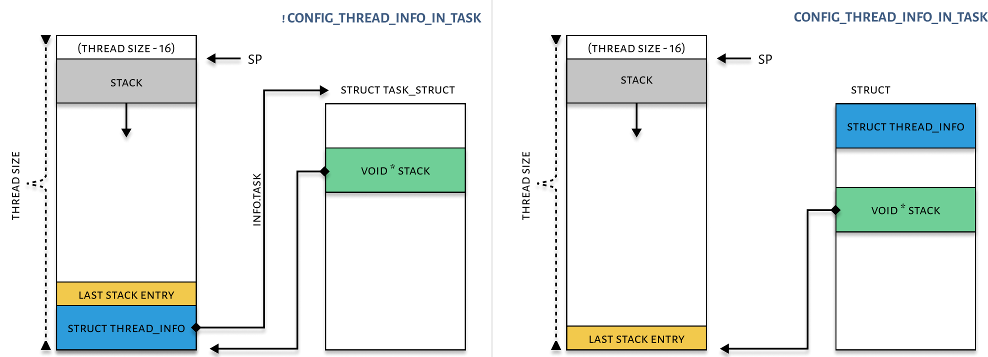


FIGURE 3.2: *Stack patching schema*

associated with the process, which can be accessed in constant time when that process is executing. When a PMI is fired, its handler can quickly access the current thread stack to determine what should be done with the generated samples.

There are two issues to consider if this strategy is adopted. First, the last entry on the thread stack might not be the very topmost memory address in the stack pages. Indeed, depending on the Linux version or the underlying hardware architecture, we might find the `struct thread_info` data structure at the end of the stack area, as shown in Figure 3.2. Therefore, care must be taken to determine the proper placement of the `struct thread_info` data structure, not to overwrite its content—this is a check that can be performed at compile-time anyhow.

The second issue to consider is related to the natural growth of the stack. Indeed, if the stack grows, our flag might be overwritten. This circumstance might not be easily detected, as this might be due to a perfectly legit memory write[5].

Some techniques can be used as mitigation. A possibility is to rely on a cyclic redundancy check (CRC) to assess before the flag is read— the probability of a false negative depends on the robustness of CRC. This solution introduces a minimal level of uncertainty that affects the system's reliability. A second possibility entails relying on dedicated hardware debug registers that are typically used to define hardware breakpoints. The firmware generates a trap every time the address is accessed—according to the debug register configuration, it may filter reads, writes, or even execution accesses. The associated trap routine should then manage the event accordingly. Relying on debug registers does not introduce concurrency issues because each

[5]This problem is one reason that led kernel developers to map the kernel stack from physical onto virtual memory and to move the `struct thread_info` data structure out of the stack, as shown in Figure 3.2.

core can leverage its own set. Nevertheless, both relying on a CRC check or on a debug register introduces an additional overhead to the simple task of determining whether a thread should be profiled or not.

The last solution is the simplest one: the `struct task_struct` can be extended with a dedicated flag to keep the monitoring state. While this solution is likely to provide the best performance, it is not viable if the monitoring agent is implemented purely via an LKM.

The second essential requirement to perform per-thread profiling is being able to intercept a context switch. As mentioned, this is fundamental because PMUs observe the CPU's execution at a system level. Still, if the scheduler swaps out a monitored thread, its performance data must be updated to allow resetting the profiling trace. Similarly to the fast IRQ case that we described in Section 3.1, the context switch kernel code is not exposed to LKMs, because it is considered part of the operating system's internals. Therefore, if our monitoring agent is implemented as a module, we must rely on additional facilities exposed by Linux to be informed of the occurrence of a context switch.

The Linux kernel is disseminated of several fixed hook points, called *tracepoints*, which can be used to install an arbitrary function (probe) to be invoked every time the tracepoint is met. This facility has been designed to help kernel developers perform debugging activities of specific portions of the kernel. Nevertheless, we propose to repurpose this facility as a generic *callback capability* that also allows modules to be informed of specific activities taking place—in our case, a context switch.

A tracepoint can be either enabled or disabled. It is enabled if at least one probe is installed in the specific tracepoint. All installed probes are queued and are called one by one when the tracepoint is reached, thus allowing to redirect the flow to custom registered functions. If a tracepoint is disabled, the introduced overhead is minimal. Tracepoints are declared via the macro `DECLARE_TRACE(name, proto, args)`, requiring as input the tracepoint's name, the prototype of the callback functions, and the parameters name. The Linux kernel already ships a tracepoint named `sched_switch`, which is triggered every time the scheduler selects the next task to be scheduled, thus allowing us to observe both the old and the new threads installed on the CPU.

In Listing 8 we present our approach to implementing in an LKM-based monitoring agent a callback function to selectively profile the system at a thread-level basis. The function `register_ctx_hook` allows registering the callback function at the selected `sched_switch` tracepoint, if it is found in the running kernel. To perform this check, we rely on the `for_each_kernel_tracepoint` macro. It iterates among all the tracepoints defined at kernel compile time, performing the check implemented in the

lookup_tracepoint function, eventually selecting the correct tracepoint reference, if present[6]. Once identified, we rely on tracepoint_probe_register to register our custom callback function ctx_hook_func.

Listing 8 Context Switch Callback based on Tracepoints.

```
1   struct hook {
2           char *name;
3           void *func;
4           struct tracepoint *tp;
5   };
6
7   static void ctx_hook_func(void *data, bool preempt, struct task_struct *prev,
8                     struct task_struct *next))
9   {
10          if (this_cpu_read(pcpu_pmcs_active) && !is_profiled(next))
11                  // ... update prev profiling data
12                  disable_pmc_on_this_cpu();
13          else if (!this_cpu_read(pcpu_pmcs_active) && is_profiled(next))
14                  enable_pmc_on_this_cpu();
15  }
16
17  static struct hook ctx_hook = { "sched_switch", ctx_hook_func, NULL };
18
19  static void lookup_tracepoint(struct tracepoint *tp, void *hook_ptr)
20  {
21          struct hook *hook = (struct hook *)hook_ptr;
22
23          if (strcmp(hook->name, tp->name) == 0)
24                  hook->tp = tp;
25  }
26
27  int register_ctx_hook(char *name, void *func)
28  {
29          /* Fill the hook's tracepoint */
30          for_each_kernel_tracepoint(lookup_tracepoint, hook);
31
32          if (!hook->tp)
33                  goto err;
34
35          if (tracepoint_probe_register(hook->tp, hook->func, NULL))
36                  goto err;
37
38          return 0;
39  err:
40          return -ENXIO;
41  }
42
43  void unregister_ctx_hook(enum hook_type type, void *func)
44  {
45          tracepoint_probe_unregister(hook->tp, hook->func, NULL);
46          tracepoint_synchronize_unregister();
47  }
```

Deregistering the callback is a simpler action but requires some care. We rely on tracepoint_probe_unregister to notify the kernel that ctx_hook_func

[6]In the proposed code example, struct hook is only a helper structure to aggregate all the information of interest.

should not be called anymore upon context switch. Nevertheless, this operation can be delayed. This delay can create a race condition if the deregistering operation is executed when the LKM is unmounted. Indeed, some CPU cores might try to invoke that function after the unloading is completed, thus causing a system crash. To prevent this possibility, we explicitly perform a call to `tracepoint_synchronize_unregister`, which ensures that all execution traces upon all CPUs are synchronised about the removal of the callback function.

In Listing 8 we also show the actual implementation of the callback function, namely `ctx_hook_func`. This function implements the logic to drive PMUs according to the profiling status of the scheduled/descheduled processes. This implementation also shows an additional optimisation related to the per-thread processing scheme we are discussing. Indeed, the profiling agent can inspect the old and new scheduled threads' profiling state to reduce the number of unnecessary MSR writes. Whenever a profiled thread replaces another profiled thread, the PMUs' configuration does not change; thus, reconfiguring the involved MSRs does not bring any gain but only introduces a cost. A PMU's configuration transition occurs only when the switch between two threads bumps into a profiling state change. To this end, we have introduced a per-CPU variable (`pcpu_pmcs_active`) that tells whether the current-core PMUs are active.

Beyond tracepoints, another technique can be used to inform the monitoring agent that a context switch is taking place. In particular, Linux can be configured to offer an additional dynamic probing subsystem, generally called *kprobes*. The kernel can install such probes in almost any code location. Typically, they are used to dynamically attach a callback function to any function's *entry point*, *return point* (a *kretprobe*), or even after the execution of every instruction. In order to inject the probe at the desired point, the kernel replaces part of the machine instruction that should trigger the callback activation with a *debug trap* instruction[7]. Once the control flow reaches the trap, a dedicated handler is activated to implement the logic associated with the management of the kprobe.

A general code instruction is instrumented by replacing that machine instruction with a trap, ideally, an `int3` assembly instruction which maps on debug exception. The handler of that exception is aware of the installed probe and carries out the additional logic before or after the code instruction is instrumented. This logic entails reconstructing the original instruction in a different memory location, invoking the associated callback function, and resuming the initial execution flow. If a kretprobe is installed, the kernel

[7]On x86 architectures, this is typically done by relying on the `int3` assembly instruction. It is a 1-byte instruction that can therefore be used to (partially) overwrite any instruction.

silently installs a kprobe associated with a special callback that installs a trampoline on the called function stack by replacing the return address. In this way, the callback can gain control after a return instruction is executed. Of course, on SMP machines, multiple executions flows can concurrently hit a kretprobe. In this case, the kernel has to track all the active instances, requiring (upon installation of the kretprobe) to define the maximum expected number of instances—if this value is too low, some probes might be missed. These sequences of operations come with some technical difficulties that we omit here for brevity. The interested reader can refer to Krishnakumar [86] and Mavinakayanahalli et al. [104] for a more comprehensive discussion of this support.

To notify the monitoring agent of the upcoming context switch, we insert a kprobe into the `finish_task_switch(struct task_struct *prev)` function, which is invoked at the end of the `schedule()` function just before the new thread gets control of the CPU. At the end of that function, the old process can be identified via the `prev` parameter, while the newly-scheduled one can be already reached via the `current` macro. We provide in Listing 9 the reference code to install a context switch callback by relying on kprobes—the `install_kprobe` function (lines 30–42). In particular, we rely on a kretprobe to be notified when the `finish_task_switch` function is returning. We have set the maximum number of expected concurrent instances to the number of active CPUs (line 5) because this is the maximum number of context switches that can occur simultaneously.

The `entry_handler` and the `handler` members are the callbacks that are installed at the beginning and return points of the `finish_task_switch`, respectively. The initial callback's goal is to intercept the parameter of `finish_task_switch` and make it visible to the `handler` function, which could not access that information otherwise. The return callback will use that value to accomplish several tasks, such as managing the PMUs' state and updating the existing process collected data.

There is an additional aspect that is relevant to discuss concerning per-thread profiling. As mentioned earlier, we might be interested in monitoring a group of threads. At any execution time, they might spawn new processes/threads, and the monitoring agent should be able to discriminate whether the new threads belong to the monitored group or not. Nevertheless, if the agent is configured to include newly spawned threads into the monitoring pool, it must be able to detect the actual creation of the new threads. A similar consideration deals with the termination of threads, particularly to enable the monitoring agent to perform cleanup tasks. For instance, if the monitored threads are kept in a hash map, their termination requires removing them from the data structure and freeing all related metadata.

We rely on an approach similar to what we have done to intercept context

Listing 9 Context Switch Callback based on Kprobes

```
struct kretprobe krp_post = { .handler = finish_task_switch_handler,
                              .entry_handler = finish_task_switch_entry_handler,
                              .data_size = sizeof(struct task_struct *),
                              .maxactive = NR_CPUS };

int finish_task_switch_entry_handler(struct kretprobe_instance *ri,
                                     struct pt_regs *regs)
{
        unsigned long *prev_address = (unsigned long *)ri->data;
        /* Take the reference to previous task */
        *prev_address = regs->di;
        return 0;
}

int finish_task_switch_handler(struct kretprobe_instance *ri,
                               struct pt_regs *regs)
{
        /* Get the reference to the leaving process */
        struct task_struct *prev =
                (struct task_struct *)*((unsigned long *)ri->data);

        if (this_cpu_read(pcpu_pmcs_active) && !is_profiled(current))
                // ... update prev profiling data
                disable_pmc_on_this_cpu();
        else if (!this_cpu_read(pcpu_pmcs_active) && is_profiled(current))
                enable_pmc_on_this_cpu();

        return 0;
}

int install_kprobe(void)
{
        int err = 0;

        /* Hook function */
        krp_post.kp.symbol_name = "finish_task_switch";

        err = register_kretprobe(&krp_post);
        if (err)
                pr_warn("Cannot hook post function - ERR_CODE: %d\n", err);

        return err;
}
```

switches. In particular, it is possible to use both tracepoints and kprobes to intercept the creation/termination of threads. For this specific goal, the static tracepoints can be retrieved by looking for `sched_process_fork` and `sched_process_exit`, respectively. Analogously, kprobe instances can be attached to the _do_fork and `do_exit` system functions. Listing 10 shows the reference implementation of the callbacks when relying on both techniques.

Listing 10 Intercepting Thread Creation/Termination Events.

```
1   // Static Tracepoints
2   void fork_hook_func(void *data, struct task_struct *parent,
3                       struct task_struct *child)
4   {
5           if (is_profiled(parent))
6                   pid_register(child);
7   }
8
9   void exit_hook_func(void *data, struct task_struct *p)
10  {
11          pid_unregister(p->pid);
12  }
13
14  // Dynamic Kprobes
15  int exit_pre_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
16  {
17          pid_unregister(current->pid);
18          return 0;
19  }
20
21  int fork_ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
22  {
23          unsigned long retval = 0;
24
25          if (is_profiled(current)) {
26                  retval = regs_return_value(regs);
27                  if (is_syscall_success(regs))
28                          /* Fork return value is the child's pid */
29                          pid_register(retval);
30          }
31          return 0;
32  }
```

3.3 Performance Data Buffering

So far, we have discussed several techniques to enable high-frequency profiling across different domains. As a result, by relying on the techniques mentioned above, a profiling agent can collect a vast amount of data potentially of interest for different optimisation strategies based, e.g., on thread-affinity, hosting cores, or execution mode. Therefore, an additional problem that a monitoring agent has to face to support online self-tuning strategies is how to manage the collected data. Several crucial aspects in this context are related to memory retention policies, concurrency in data structures access, cache pollution, and overhead in general. This section presents different strategies and data structures that a monitoring agent can employ to deal with the (possibly huge) amount of data collected at runtime from PMUs.

Independently of the buffering strategy, memory management is an essential aspect to consider. Indeed, the data placed in the buffers are generated while running the PMI code, according to some of the strategies discussed above. Dealing with memory allocation while running in an interrupt context is tricky because of constraints that the memory allocator

should consider. In particular, if the PMI handler has to allocate the buffers to keep performance samples, the allocation can only be done via `kmalloc`, setting the `GFP_ATOMIC` flag—other kernel-level allocators, such as `vmalloc`, cannot be used in interrupt context, because they may make the invoking thread sleep. Conversely, a `kmalloc` call may fail under high memory pressure, leading to a possible sample loss.

The implicit strategy in all buffer management techniques described in this section is to rely on pre-allocation, possibly when installing the profiling agent. Nevertheless, we note that a pre-allocated buffer may get filled, a situation that brings the same level of complexity if dealt with in an interrupt context. As we will show, a viable solution is to rely on buffering data structures that may overwrite older samples if the buffer is filled. This approach simplifies buffer memory management but may render a self-tuning system unreliable. Therefore, if sample loss cannot be tolerated, the profiling agent should be configured with a sample generation frequency and a buffer size such that the likelihood for a buffer to become full is extremely low.

3.3.1 System-wide Buffering

The simplest scenario is when the monitoring agent has to collect system-wide performance data. This scenario can be helpful when, for example, the data gathered by the agent are used to optimise some global action, such as the scheduling of routine housekeeping operations, which do not depend on the behaviour of specific threads or applications.

A straightforward buffering strategy is to define a single global buffer to store all the generated information during system monitoring. Nevertheless, since PMU data are generated at a per-core level, this shared global buffer must be synchronised. Moreover, accesses from the monitoring agent to deliver performance data to userspace must be synchronised to avoid incorrect reads while write operations occur. Synchronisation can quickly become the performance bottleneck given the high concurrency we expect to generate performance data.

We propose a candidate data structure that can be used to reduce synchronisation costs in case of global buffering of performance data, namely a non-blocking [63] Single-Writer/Multiple-Readers circular queue. In this data structure, writes have higher priority over reads, explicitly because we account for the scenario in which data generation takes place within the PMI, whose duration should be kept as low as possible. Rather than relying on locks, we depend on Read-Modify-Write (RMW) atomic instructions to enable a fine-grain synchronisation of read/write operations. In particular, the proposed algorithms to manipulate the data structure relies on atomic `fetch_and_add` (FAA) and `compare_and_swap` (CAS) instructions, which

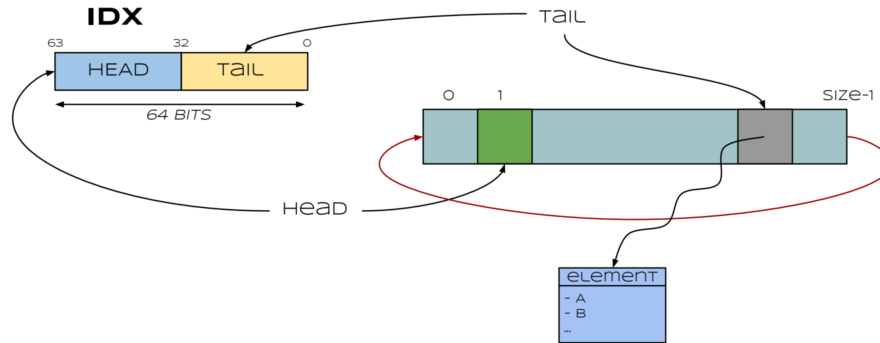


FIGURE 3.3: *General Organisation of the Non-Blocking Circular Queue.*

are available on most off-the-shelf architectures. FAA takes a memory address and a value. It first reads the content at the given address, then adds the passed value to the content of the memory location, updating the memory content while returning the old value. All these operations are executed atomically. Conversely, CAS accepts a memory location and two values: an *expected* old one and a new one. It updates the memory location with the new value only if the expected old value is currently stored at the given memory address. Again, all these operations are carried out atomically.

Figure 3.3 depicts the organisation of our non-blocking circular queue. The actual buffer keeping performance data is a contiguous array, while the current state of the queue is maintained by relying on two indices to identify the HEAD and TAIL entries in the array. The former indicates the position where someone can start reading, while the latter is the index where a new element can be stored. The tail and head indices are stored in the same 64-bit variable (named IDX), the largest word that can be manipulated atomically using RMW instructions on 64-bit x64 architectures. This choice allows us to reduce the number of memory accesses when performing operations on the queue. The least-significant 32 bits of IDX keep the tail index, while the other half holds the head. The buffer follows a First-In-First-Out (FIFO) behaviour and can be manipulated with two methods: INSERT() and REMOVE(), for queue elements enqueue and dequeue, respectively. The queue can be in one of the following states, as depicted in Figure 3.4:

- EMPTY: head and tail are equals. Only INSERT() can be performed.
- FULL: head and tail are logically equal, but the tail is one cycle[8] forward. Both INSERT() and REMOVE() can be performed, but the

[8]The circular logic implies that an index logically ranges from 0 to length - 1, but its real value may only increment. A cycle represents one walk through the entire buffer. For the tail, one cycle forward means that REAL(TAIL) = REAL(HEAD) + length.

former will overwrite the oldest element.

- NORMAL: head and tail are different[9], and both operations can be called.

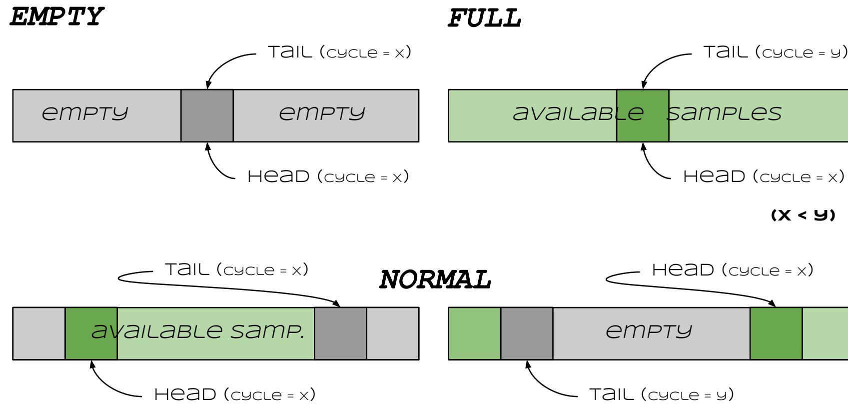


FIGURE 3.4: *Possible Queue States.*

We report in Algorithm 3.1 the pseudocode for the INSERT() operation. As mentioned, it is used within a PMI, so it has been designed to offer a wait-free progress guarantee, which is achieved by relying on a replacement strategy of the samples in case the buffer is full. This means that to rely on this data structure correctly, either the profiling application can tolerate some sample loss, or the frequency at which samples are generated must be fine-tuned to avoid incurring in overwrites.

An INSERT() reads IDX to check if the queue is in the FULL state (line 2). If so, it tries to perform a CAS to grow the tail index (line 3). The CAS is necessary to guarantee atomicity between the read and the increment. Two cases are possible:

- Success: the tail is atomically incremented so that any reader is potentially blocked (see the remove() algorithm in Algorithm 3.2). Then the new element is written on top of the oldest element (line 4). Finally, the head index is incremented so that reads can be executed (line 5).
- Failure: a reader concurrently dequeued an element. Therefore, the queue is back in the NORMAL state. The result of the initial check is no longer valid.

[9]The logic value of head can be greater or smaller than the tail, depending on the respective cycles.

Algorithm 3.1 Enqueue Operation (Single Writer)

```
1: procedure INSERT( )
2:     midx ← IDX
3:     if HEAD(midx) + SIZE = TAIL(midx) then
4:         if CAS(IDX, midx, TAIL(midx) + 1) then
5:             WRITE(elem, TAIL(midx))
6:             IDX ← HEAD(IDX) + 1
7:             return
8:         end if
9:     end if
10:    WRITE(elem, TAIL(midx))
11:    FAA(IDX, midx, TAIL(midx) + 1)
12: end procedure
```

If the initial check did not succeed, we are in either the NORMAL or EMPTY state. Thus, the writer can store the new element (line 9) and then advance the tail to notify all readers that a sample is available (line 10). This increment must be done via an FAA operation because the head and tail are both stored in IDX, and readers' and the writer's concurrent accesses may occur.

Algorithm 3.2 shows the pseudocode for the dequeue operation. This operation is not expected to be executed in an interrupt context, so it can rely on retry loops in case of failures, making it a non-blocking operation. A reader must wait if the queue is FULL and the tail is a step forward (lines 4–6) because the writer is performing a concurrent overwrite of the element at the head index. Then, the reader checks whether the queue is EMPTY, in which case the operation returns an empty value (lines 7–9). If the queue is in the NORMAL or FULL state, the operation shall return the first available element (line 10). An additional CAS is used to mark the element as consumed by updating the head index (lines 11–13). Similarly to the discussion on Algorithm 3.1, relying on a CAS operation on IDX ensures that if a writer has changed the queue state concurrently to the read, the reader is forced to observe again the updated state thanks to the CAS failure. As mentioned, this implementation gives higher precedence to write operations.

3.3.2 Per-thread Buffering

A different buffering strategy is related to keeping all performance data separated per thread. As discussed in Section 3.2, it is possible to let the monitoring agent associate PMU data to a specific thread. Keeping the

Algorithm 3.2 Dequeue Operation (Multiple Readers)

```
1: procedure REMOVE( )
2:     loop
3:         midx ← IDX
4:         while HEAD(midx) + SIZE < TAIL(midx) do
5:             midx ← IDX
6:         end while
7:         if HEAD(midx) = TAIL(midx) then
8:             return empty
9:         end if
10:        elem ← READ(TAIL(midx)) ▷ Make a local copy of the element.
11:        if CAS(IDX, midx, HEAD(midx) + 1) then
12:            return elem
13:        end if
14:    end loop
15: end procedure
```

obtained data separated can support many online optimisation strategies, e.g. in scenarios where thread placement on the CPU cores tailors a reduced contention effect on the caching subsystem [27] or to improve memory access latency in NUMA systems [41].

Keeping per-thread data can be done by relying on multiple strategies. Clearly, the non-blocking buffer presented in Section 3.3.1 can be immediately repurposed. Indeed, it is sufficient to instantiate multiple queues, one for each thread, and use it to store the data. Associating the thread with the relevant queue is also a straightforward task. As discussed in Section 3.2, it is already necessary to rely on some data structure (e.g., a hash map) to identify whether a thread is being profiled or not—in which case, we have to keep sampled data in the dedicated queue. Any data structure used for this purpose can be augmented with a pointer allowing to locate in memory the per-thread queue. We note that this strategy is also compliant with thread registration/deregistration because the queues can be efficiently allocated/deallocated.

A second strategy is again bound to the perf_events subsystem introduced in Section 3.2. As mentioned, its integration in the kernel is such that some data structure already keep members used to support its execution. Referring again to Listing 7, we have shown that shows an example of how to register, deregister and query a perf event. Once the perf event is instantiated, custom information is attached to `event->pmu_private`, then the event is queued into the dedicated per-process perf event list. To check whether a process should be monitored, the monitoring agent can search

for the dummy event in the corresponding list and check the related private_data. Even though all the operations are kept consistent through a mutex, our experience shows that no significant overhead is introduced.

A third strategy entails directly modifying kernel sources. In this case, a direct modification of the `task_struct` allows to include a pointer to the data buffer directly. However, this approach requires to deal also with process creation and termination to allocate/release the buffer. This latter point requires some care because concurrent accesses to the buffer should be made consistent also with respect to memory reclaim. Given the intrinsic parallel nature of many workloads, protecting buffer access for performance data reading through, e.g., locks can provide a significant performance penalty, as we have already discussed. Moreover, in highly concurrent applications, per-thread buffering might require allocating a considerable number of buffers, generating higher memory pressure if the thread count of the application is significantly high.

3.3.3 Per-core buffer

Taking into account the design of the PMU in x86 systems, commonly, this dedicated circuitry can observe *in-core* executions or machine-level influences, namely the *off-core* events. The latter can be programmed and consequently read by any CPU core. An analyzer tool may employ the off-core PMUs when approaching a comprehensive assessment of the external components, such as measuring the memory bandwidth or latency of the main memory.

Another buffering strategy deals with storing performance data on a per-core basis. This is probably the most straightforward way to deal with PMU data because each CPU core is intrinsically responsible for its dedicated PMUs, which provide data related to the events happening on that core[10]. This buffering strategy can be leveraged in scenarios where the effective usage of hardware resources can drive thread placement on a per-core basis [43].

Given the per-core private nature of PMUs, installing a dedicated buffer for each CPU core requires synchronising only read/write accesses to the buffer. This can be achieved by relying on the non-blocking queue described above or on more classical lock-based synchronisation strategies, possibly employing read/write locks. In both cases, a per-CPU variable can be used to maintain a reference to the sample buffer.

[10] There are some exceptions when Simultaneous Multi-Threading is enabled. Logical threads share the PMUs of the physical core. The resources are then equally divided, but many events may refer to the overall physical core activity rather than the logical processor.

Per-core buffering can provide several advantages also at the micro-architectural level. For example, this strategy can improve cache exploitation because the same buffer is used across context switches, thus allowing the cache prefetcher to optimise memory accesses.

Finally, it is noteworthy that per-core buffering cannot be effectively used to support profiling strategies different from per-core ones. Indeed, one might think that this lower-overhead solution might also be used to perform, e.g., per-thread profiling by post-processing the buffers to extract per-thread data. Unfortunately, in this case, the incurred overhead can be higher than relying on some of the aforementioned buffering solutions. Indeed, the data in the buffer should be demultiplexed depending on the associated thread, which could be costly. Moreover, a thread may be migrated to a different core upon a context switch, thus scattering all the data of interest across all per-core buffers. Considering our online profiling goal, the overhead of this strategy can be too high.

3.3.4 Accessing Profiling Data from User Space

While a kernel-level facility is fundamental to implementing a profiling agent based on PMU data, a modular organisation of an autonomic self-tuning system would benefit from separating the performance data collection part from the autonomic logic. In particular, it may be desirable to run the latter in user space, both for maintainability and performance reasons.

In this scenario, it is fundamental to allow userspace applications to read performance samples. A first strategy is to directly share the buffers by relying on a dedicated mapping. This configuration is easily achievable by employing the `mmap` system call. On the one hand, providing the user-level applications with unmediated access makes them faster than those exploiting the conventional system call interface. Moreover, creating a new mapping in the process' virtual address space for a fixed memory area minimises, after some accesses, the cost of performing the mapping itself[11].

At the same time, this approach shows several drawbacks. First, the userspace application must be aware of the data buffer format—for example, if the aforementioned non-blocking queue is used, its access must be consistent with the proposed algorithms. Moreover, userspace applications must synchronise their accesses with kernel-level code to enforce consistency. Overall, this approach requires a coupling logic between the user program and the underlying kernel, which can be undesirable due to higher complex-

[11]Completing the memory mapping requires adjusting the page-table entries for the calling process. Consequently, exploiting this approach to read memory mapped locations just once is not optimal and may introduce an extra overhead compared to `read` or `write` system calls.

ity. Moreover, when dealing with a high-frequency data generation as we do, the user application may not be fast enough to consume them before the circular buffer becomes full and the data start being overwritten. In this kernel/user interaction, the problem is exacerbated because scheduling policies might arbitrarily delay the activation of the userspace application. In this sense, finding an optimal data generation frequency and buffer size may be impossible.

A second strategy is to leverage standard virtual file system (VFS) capabilities to build an interface to access the buffers in a mediated way. In particular, it is possible to install in the system a set of pseudo-files that allow, via standard `open`, `read`, `write`, `seek` and `close` system calls, to configure the kernel-level activity and retrieve copies of the data from the buffers. This solution decouples the internal management of PMUs from the "presentation layer", allowing for a more flexible implementation of the profiling agent. Nevertheless, requiring userspace applications to interact with a linear file explicitly can be more complex than needed. In fact, a typical approach to consuming performance data is to read them only once, to perform some computation (e.g., supporting the autonomic self-tuning strategy).

Therefore, while we propose anyhow to rely on the `proc` file system, we highlight a different strategy. First, we install a set of pseudo-files that allow reading all possible combinations of data buffering described above. Second, we exploit the read-once nature of performance samples to provide a simple API that can deliver a requested number of performance samples to userspace in an iterator-like fashion. A `struct seq_operations` contains the reference to the `seq_file` operations: `start`, `stop`, `next`, and `show`. Those functions implement the core of the `seq_file` iterator-like navigation. Of course, as we will show experimentally, this approach is slightly more costly than the `mmap`-based solution due to the different layers that compose the VFS.

3.4 Experimental Assessment

This section presents an experimental assessment of the performance implications of the different techniques discussed in this chapter. All tests have been executed on Ubuntu 20.04 LTS with Linux Kernel 5.4.127 running on a machine equipped with an i7-10750H 6x (SMT) and 16Gb of Ram.

We have used the stress-ng benchmark suite [82], which comprises a massive set of stress tests known as *stressors*. Each of them is designed to target a specific component or subsystem, both at the hardware and software levels. The set of stressors that we have selected is representative of different kinds of workloads. This selection allows us to study the impact

on the performance of our proposed strategies in various contexts, i.e. when considering CPU-bound, IO-bound, or syscall-intensive applications. The selected stressors are:

- *cpu*: a CPU-bound application that starts n different workers that iteratively run heavy-computation functions;
- *cache*: a memory-intensive application that starts n workers performing widespread random memory read and writes to thrash the CPU cache;
- *memcpy*: n workers that copy 2 MB of data from a shared region to a buffer using `memcpy` and then move the data in the buffer with `memmove` with different alignments;
- *atomic*: n workers that exercise various `__atomic_*()` built-in operations on 8, 16, 32 and 64-bit integers shared among the workers;
- *matrix-3d*: a CPU-intensive benchmark where n workers perform matrix operations on floating-point values;
- *bsearch*: perform a binary search on a 32-bit integer sorted array, using n workers;
- *fork*: a syscall-intensive benchmark where n workers continuously fork children that execute stress-ng and then exit almost immediately;
- *switch*: a benchmark using n workers that send messages via a pipe to a child to force context switching;
- *clone*: n workers that create clones via the `clone` system call.

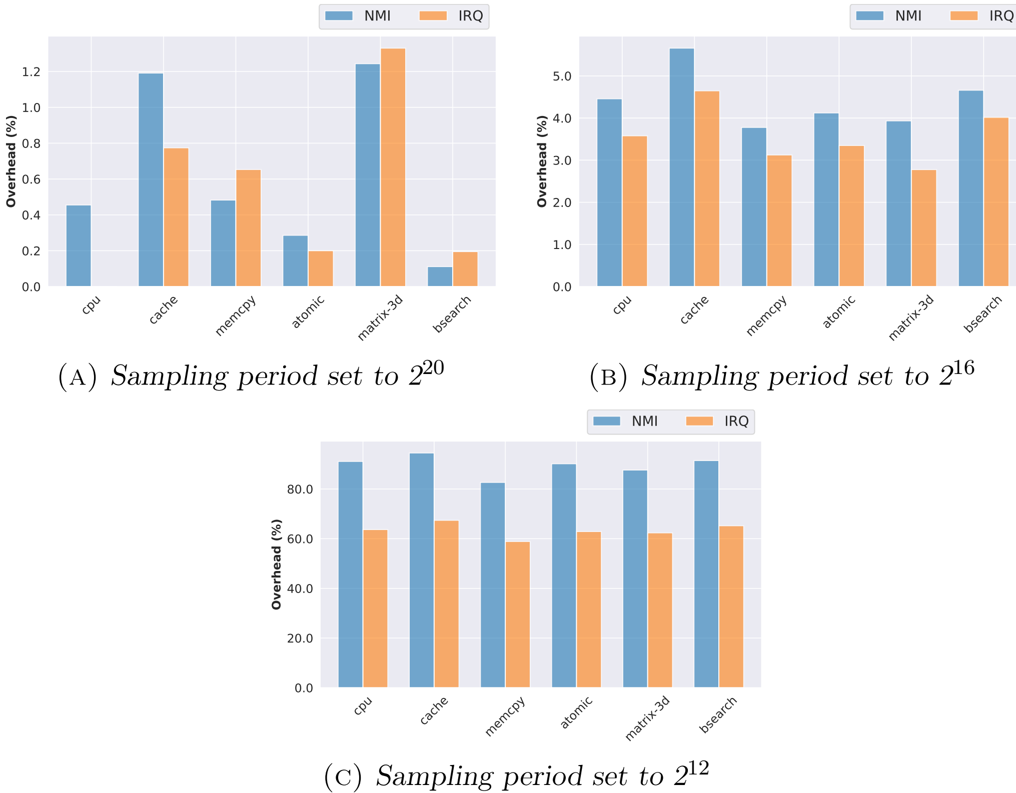
The reference implementation that we have used for our experimental assessment is based on an LKM which implements all the components discussed in this chapter. The module is based on a set of weak-symbol functions implementing the different subsystems of the monitoring agents (e.g., context switch hooking or buffer management) that are overridden each time a specific implementation is under test. By relying on this organisation, the overall code path is kept relatively stable across the different tests, thus enabling us to compare the performance results reliably.

Conversely, some configuration variables are exposed as module parameters to simplify the configuration. Some are the *sampling period* (defined in the elapsed clock cycles domain) and the *PMI vector line*. Beyond the capabilities discussed in Section 3.3.4, we have introduced proc files to access and query tool parameters such as the number of collected samples. All the results are averaged over 5 different runs.

3.4.1 Efficient Collection of PMU Data

For this experiment, we have relied on a patched Linux Kernel (version 5.4.127) to expose to our LKM the API for fast IRQ registration, as de-

scribed in Section 3.1. This experiment compares the overhead of generating and processing PMIs when relying on the fast IRQ mechanism or NMI lines. To assess the accuracy of the different strategies, we also compare the amount of data collected by varying the sampling period.



(A) *Sampling period set to* 2^{20}

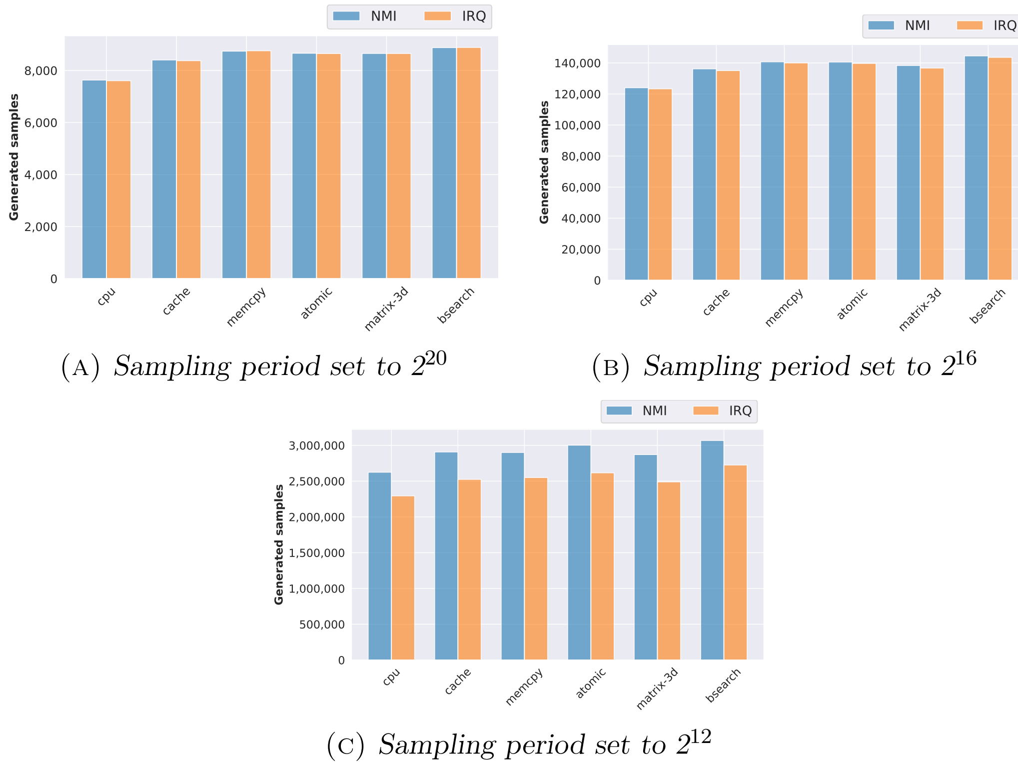
(B) *Sampling period set to* 2^{16}

(C) *Sampling period set to* 2^{12}

FIGURE 3.5: *Workload overhead comparison between IRQ- and NMI-based PMIs.*

Figure 3.5 reports the overhead increment of both solutions compared to an execution where no sampling method is enabled. As seen in Figure 3.5a, a larger sampling period does not make any of the two solutions outperform the other entirely. It is an expected result, as this low-frequency generation of samples is unlikely to impact the overall system activities. Conversely, it is clear that the fast-IRQ mechanism outperforms the NMI-based solution for higher-generation rates.

We also monitored the number of generated samples for both configurations, reported in Figure 3.6. For lower frequencies, both techniques collected a comparable amount of samples. However, at higher rates (see Figure 3.6c), the NMI-based solution reports a higher number of samples—around 20% more than the IRQ-based counterpart. This higher count is due to Linux's NMI handling, which cannot prevent PMUs from starting their



(A) *Sampling period set to* 2^{20}

(B) *Sampling period set to* 2^{16}

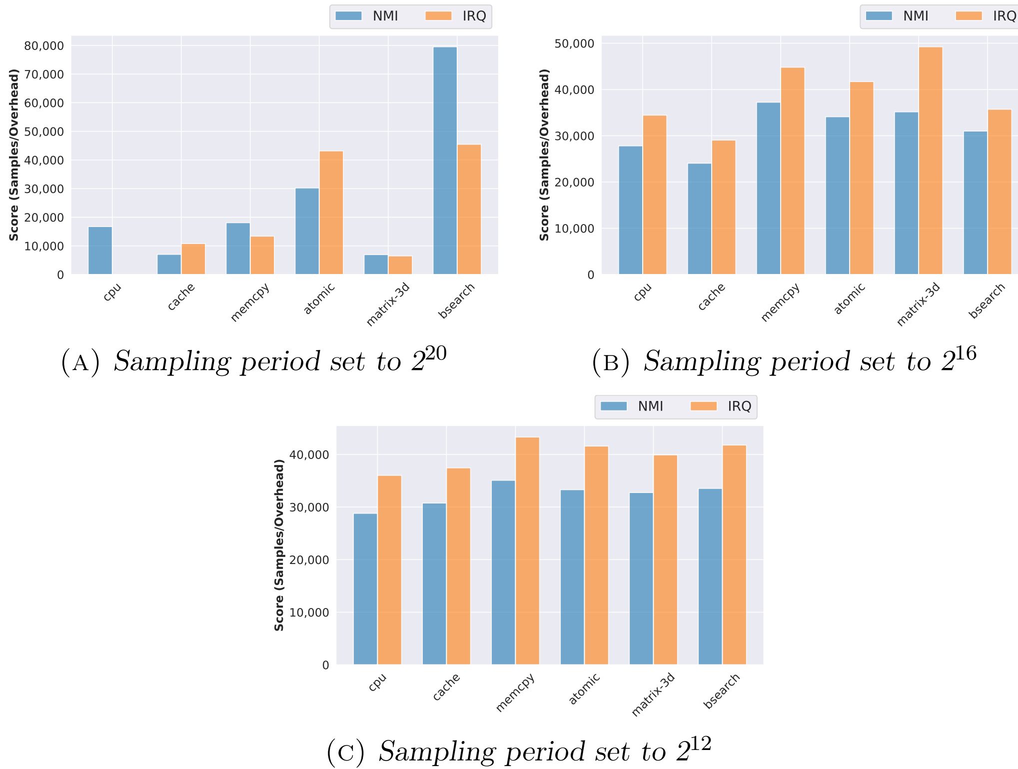
(C) *Sampling period set to* 2^{12}

FIGURE 3.6: *Samples generation comparison between IRQ- and NMI-based PMIs.*

monitoring before leaving the PMI routine[12]. Therefore, the higher count is mostly related to the PMI routine monitoring itself, which is a clear form of Heisenmonitoring [113].

Figure 3.7 provides an additional perspective on this result. In particular, we show an *efficiency score* value, which captures how many samples are generated per overhead unit. These results confirm that the fast IRQ-based solution is the most effective at medium-to-high sample generation frequencies. Overall, this experiment clearly indicates that the traditional NMI-based approach used in the literature cannot be effectively used if online profiling activities are the goal of PMU exploitation unless the monitoring agent is interested in a coarse-grain profile of the application. In this latter case, it can employ both strategies.

[12]This issue has also been recognized by hardware vendors. In the latest x86 processor models, a dedicated MSR allows freezing PMU operations during PMI handling to avoid data pollution. Unfortunately, the Linux NMI handler is implemented in such a way that the interrupt context configures the system to execute the required handler, then performs an *iret* instruction to process the handler of the interrupt context, thus *defrozen* the HPCs.



(A) *Sampling period set to* 2^{20}

(B) *Sampling period set to* 2^{16}

(C) *Sampling period set to* 2^{12}

FIGURE 3.7: *Efficiency between IRQ- and NMI-based PMIs.*

3.4.2 Discriminating the Profiling Domain

In this experiment, we evaluate the performance implications of the techniques proposed in Section 3.2 to discriminate the profiling domain. We first focus on the overhead associated with tracepoints and kprobes. For this purpose, we have selected from the stress-ng suite the *switch*, *fork*, and *clone* stressors, that exercise high pressure on thread creation and context switches. The system activity is kept as high as possible by instantiating two processes on each CPU core and executing them simultaneously. Figure 3.8 shows the overhead when relying on the two strategies. The results are expected. The more complex architecture of kprobes shows a higher overhead, mainly when many context switches are observed.

We then consider the different per-thread filtering techniques, which combine the ability to mark a process for profiling activity and create room to store per-process data. Such dedicated memory may store profiling metadata (e.g., PMUs state upon context switch) or collected information (e.g., collected data samples). We set up the environment to execute a single instance of each stressor first (*SINGLE* in the plots), and a concurrent version spawning two processes per each available CPU core (*PARALLEL* in

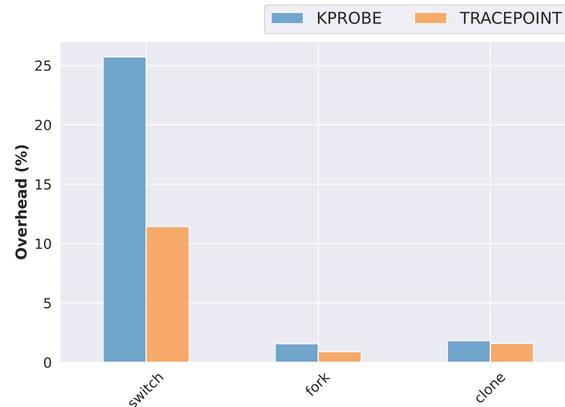


FIGURE 3.8: *Runtime overhead of context-switch intensive workloads with active system hooks.*

the plots). In this experiment, the sampling sampling period has been set to 2^{14}, while the PMI management is based on NMIs.

Figure 3.9 reports the overhead for both configurations. From the results, the hash table strategy incurs a higher performance penalty. This result is expected as the global instance is concurrently accessed by all the CPU cores and enforces consistency thanks to coarse-grain locking primitives. The high-frequency PMI firing rate amplifies the pressure on the synchronisation mechanism that, in the PARALLEL configuration, shows an even higher experienced overhead.

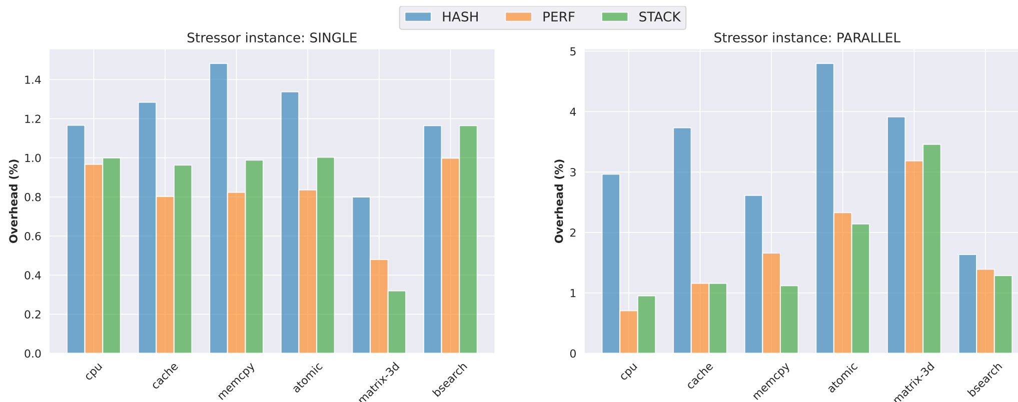


FIGURE 3.9: *Comparison among different approaches to provide per-process private data.*

We have then focused on the different buffering strategies. In this experiment, whose results are reported in Figure 3.9, we assessed the overhead of each discussed method. The results show that the non-blocking queue incurs the highest overhead in most configurations because coarse-grain lock-

ing mechanisms protect its consistency. The *per-CPU* solution introduces the most negligible overhead, just followed by the *per-thread* implementation. The difference between the two is due to the improved data locality of the former, which continuously accesses the same local buffer instance.

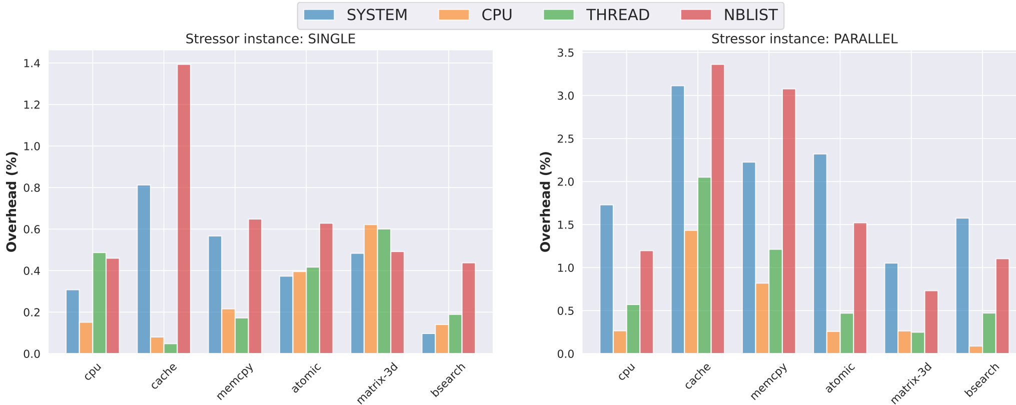


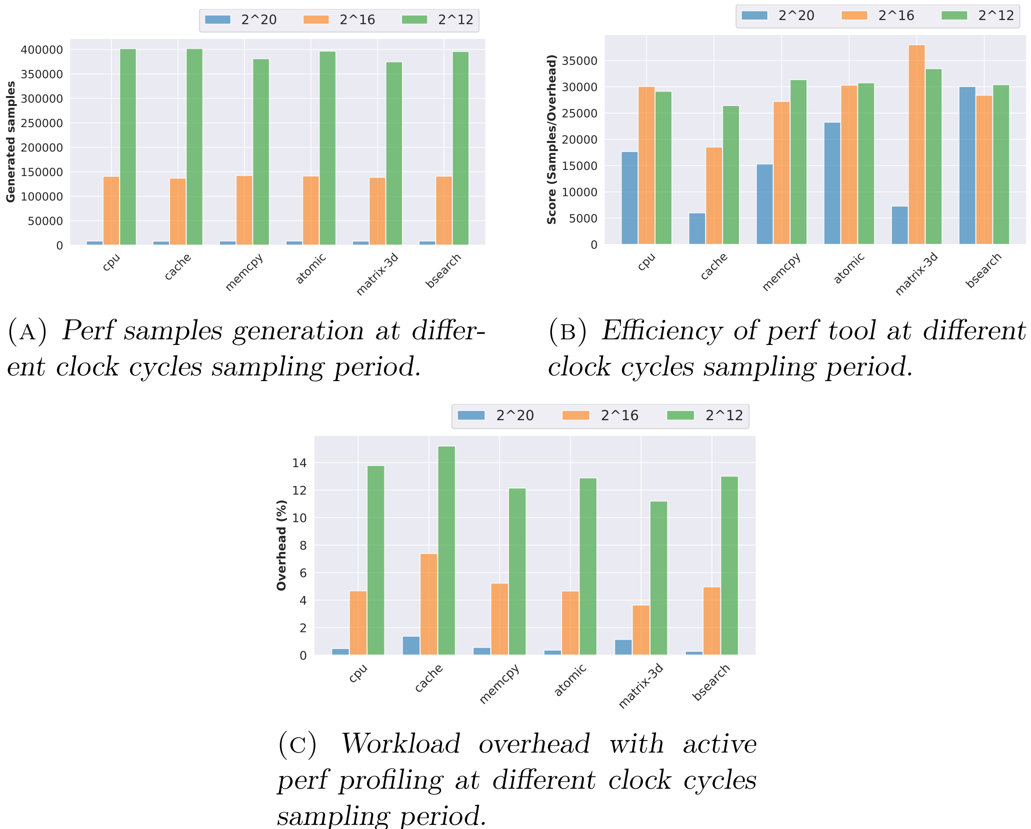
FIGURE 3.10: *Comparison among different buffering policies to collect process profiling data.*

Conversely, switching the buffer reference upon a context switch negatively affects cache performance. Such an effect is more noticeable in the right plot, where the number of participating processes demands an intense scheduling activity. Analogously, the drawbacks of the global buffer instance (referred to as *SYSTEM* in the plot) can be observed when the writers increase and the synchronisation mechanism comes in place. We recall that test set the sampling period to 2^{12} generating a high number of writing requests. Finally, we reviewed the non-blocking list implementation by setting the per-CPU policy. This solution, as we can observe, significantly impacts the workload runtimes. The most affected stressors are *cache* and *memcpy*, which, as part of their goal, concentrate their activity on the cache subsystem. This phenomenon originated from the internal implementation of non-blocking operations and interconnected hardware bus locking. Conversely, our non-blocking queue guarantees the writer's priority to preempt a reading action. The underneath fine-grain synchronisation is essential to avoid locking the CPU in the interrupt context, which may lock the processor until the read completion.

We have also carried out a comparison with the built-in perf tool. This experiment exploits the traditional userspace interface that directly bridges the user commands and the kernel level subsystem. Perf driver leverages two buffers at the kernel level. First, a direct ring buffer is used for caching results as soon as they are ready, and then the second buffer is filled when appropriate. The userspace tool lingers in a waiting state and does not consume system resources until the buffer is nearly full. In that case, to

avoid sample loss, it wakes up and drains all the collected data. To move as close as possible to the setup of our tools, we manually defined the hardware events to be monitored and enabled the PMI on the built-in *cycles* event, which maps to the *clock cycles* hardware event. We recall that the standard PMI routine (operated by perf) is mapped to the NMI vector line. Moreover, perf contains an *anti-trashing* detector that reduces the profiling frequency rate if the interrupt routine experiences a delay over a defined threshold. However, this mechanism introduces a degree of indeterminism that cannot be easily quantified.

Similarly to the IRQ/NMI test, we evaluated the overhead (Figure 3.11) and the number of generated samples (Figure 3.11a). We can observe that the perf solution introduces a lower cost at the highest frequency rate compared to the previous results. Nevertheless, the number of generated samples is way smaller at the fastest frequency due to the auto-tuning frequency policy. Figure 3.11b shows the efficiency of the perf solution computed as the quantity of generated data over the overhead execution percentage.



(A) *Perf samples generation at different clock cycles sampling period.*

(B) *Efficiency of perf tool at different clock cycles sampling period.*

(C) *Workload overhead with active perf profiling at different clock cycles sampling period.*

FIGURE 3.11: *Perf tool evaluation at different sampling periods and 8 hardware events monitored*

3.4.3 Accessing Profiling Data from User Space

In this last experiment, we evaluate the overhead associated with the different techniques explored in Section 3.3.4 to retrieve performance data from kernel space. We evaluated their performance by reading a large buffer allocated within a kernel module and accessing it through the different implemented interfaces. Moreover, each test has been run by varying the read identified as *stride*, namely the amount of data accessed within every single request.

FIGURE 3.12: *Time to fully read kernel level data via mmap- and VFS-based solutions.*

As shown in Figure 3.12, the VFS solution based on the seq_file mechanism is the least efficient in acquiring a large amount of data, despite it being the easiest to be used by typical applications. The raw VFS read (i.e., when pseudo-files give direct access to performance buffers) mitigates the overhead but still suffers from the intrinsic cost of performing a system call for each request. Indeed, the larger the stride, the smaller the number of required system calls. Conversely, as expected, the mmap-based interface offers the best performance, despite slightly reducing its effectiveness for larger stride sizes due to the higher probability of experiencing a page fault. Of course, in real scenarios, mmap-based solutions require offloading to userspace a part of the synchronisation logic, making the overall implementation more complex and coupled between kernel space and userspace.

CHAPTER 4

Online activity tracing for system security

Historically, the memory hierarchy has been introduced and equipped with multiple caches to hide the memory's high latency and overcome the performance gap between processors and memory. This component is fundamental performance-wise and has been integrated into all modern processing units. Nevertheless, attackers have repeatedly abused it to extract information from the operating system (OS) kernel or victim processes, circumventing the process confinement enforced by standard modern operating systems.

These kinds of attacks are commonly and generically referred to as *cache-based side-channel attacks*, and many different techniques have been proposed in the literature [65, 161, 93, 44, 92, 84]. They rely on observing non-functional properties of the cache architecture (e.g., timing the execution of cache-accessing operations) to infer information on a victim process or to leak data. These attacks have also been beneficial to extract data when side effects on the cache architecture are generated by exploiting transient execution CPU vulnerabilities. To exploit these vulnerabilities, the attacker relies on some speculative execution facility from the underlying computer architecture to load into the cache architecture some data from the underlying OS kernel or from a different userspace application whose access is prevented by traditional security mechanisms, such as paging isolation.

The applications of side-channel attacks are vast. For example, they have been used to extract secret keys from cryptographic algorithms including AES [79, 116, 147] or El-Gamal [93], to steal information from the underlying operating system [92, 84], to bypass Kernel Address Space Layout Randomization (KASLR) [95], or to extract cross-VM information [148, 157, 25, 28].

The idea of using a cache-based side channel to trace the execution of a program or to leak information is not new, with the first proposals dating back to two decades ago [116, 1, 24, 166]. Despite this, the fundamental mechanism has been preserved over time, i.e., observing traces left in the caching subsystem of the computer architecture to extract information. In

this chapter, we concentrate on detecting an application mounting a side channel to indicate that an information extraction might be taking place. As we will show experimentally, this focus allows us to detect also transient execution attacks if they rely on side channels to extract leaked information.

To reduce the impact of attacks that exploit cache side channels, especially those based on CPU transient execution—namely to reduce the amount of information that an attacker can leak —several mitigation strategies have been proposed, both at the software and at the hardware level. The most notable ones are Kernel Page Table Isolation (KPTI) [55], retpolines [77], `swapgs` fences, or PCID. Notably, some of these patches (e.g., KPTI) induce a non-negligible performance drop under specific workloads, which has been estimated as high as 30% [59]. This overhead could be deemed too high in specific scenarios—examples are virtualized environments supporting 5G communications [25] and high-performance computing-oriented setups. In these scenarios, a good tradeoff could be to selectively enable security patches (either the existing ones or newly devised ones) at runtime only when software suspected to try to exploit cache side channels is detected.

To detect an application mounting a side channel with a reduced performance impact, we propose to leverage hardware capabilities offered by off-the-shelf CPUs. Furthermore, given the tight connection with the measurement capabilities of HPCs and the baseline techniques used to extract information using a cache side channel, we could assert that HPCs are the perfect candidates to build a detection mechanism for this kind of attack. While several works in the literature have followed this path, a recent result has argued that micro-architectural level information obtained from HPCs cannot distinguish between benignware and malware [167]. Similarly, another work has illustrated why many of the results in the literature cannot be considered reliable [38].

In this chapter, we come back to this problem and try to capture some common features which can be used to define *detection metrics* based on measurements obtained through HPCs. We use these metrics to detect whether a process running in the system is carrying out an attack—independently of whether the attack is carried out after having exploited some transient execution CPU vulnerability.

Our detection mechanism is system-wide. In this sense, we do not make any assumption on which process is the attacker and which is the victim. We also directly account for scenarios where multiple linked processes are used to mount the attack (e.g., relying on the `fork()` system call). We exploit the information gathered at runtime to deem some processes as *suspected*. In more detail, we concentrate on detecting the usage of side channels to extract information during an attack to indicate the *possibility* that the

process is malicious.

We explicitly acknowledge that our detection mechanism is fallible due to the degree of uncertainty associated with this kind of mechanism. Therefore, we do not take any destructive action with respect to the running process. Instead, we couple our detection capabilities with mitigation actions. We propose different mitigation actions automatically enforced by the operating system as soon as a process is suspected as malicious. They entail a limitation in the scheduler freedom at deciding what CPU resources should be assigned to some process or the selective (per-process) activation at runtime of security patches against transient execution vulnerabilities.

To reduce the incidence of false positives and negatives, we rely on a self-adjustable observation window coupled with a scoring system. This approach is meant to reduce the probability that benignware with pressure on the memory hierarchy is suspected or to increase the likelihood to suspect processes that perform many non-malicious actions before carrying out the attack.

Our detection mechanism and the mitigations mentioned above have been implemented at kernel-level in Linux and have been exercised on multiple processors of the x86 family. We have used our patched kernel for a month, also in daily usage [1] —the patched kernel has also been used while typesetting this paper. No false-negative has been observed under that daily usage workload. Of course, this is not a guarantee that our approach could be used to enforce more intrusive policies for suspected processes like, e.g., killing suspected processes. Instead, it is an indication of the viability of using HPCs as building blocks for articulated detection mechanisms and for devising strategies where the setup of security-oriented patches can be put in place on a dynamic and per-process basis—rather than paying the cost of these patches by default when any process is active. Our reference implementation is released as open-source software.

We finally compare the performance penalty introduced in the system by these different mitigation strategies relying on standard benchmarks for operating systems [88].

The original contributions of this chapter can be summarized as:

- We introduce a practical mechanism to build metrics from measurements obtained from HPCs to detect that some cache side channel is currently being used;
- We propose an observation window and a scoring system to reduce false positives and negatives when considering a process as suspected;

[1] A video demonstration of the operations of our detection mechanism is available at `https://youtu.be/XGQ4TuqtTAI`.

- We propose a system-wide detection approach for userspace applications, which makes no assumption on which process is the victim and which is the attacker;
- We enable per-process or per-CPU mitigation strategies;
- We describe a reference implementation in the Linux kernel;
- We show the effects on performance of our proposal on multiple generations of Intel CPUs, using standard benchmarks.

4.1 Related Work

We can relate our proposal to two different families of countermeasures to attacks, namely *detection* and *prevention*. On the detection side, using HPCs is not a new idea. Many works in the literature have relied on HPCs for this purpose, e.g., for exploit detection [164, 163, 168, 145], malware detection [40, 151, 80, 53, 119, 125], firmware verification [152, 153], integrity checking [99, 26], or vulnerability analysis [34]. Unlike our proposal, these works mainly cope with attacks not explicitly oriented to cache side channels, like ROP (or more generally control flow tampering) or similarity-based malware detection.

We share the goals with a set of works that rely on HPCs to detect side-channel attacks [103, 19, 114, 120, 30, 73]. In general, these approaches rely on machine learning mechanisms, concentrate on specific attacks, do not support system-wide detection, require to know beforehand what the attacking process is, or do not consider the possibility of relying on selectively activated software patches. These are all major differences from our proposal.

On the prevention side, an important mitigation for the Meltdown attack is KPTI [55]. When running in user mode, this mitigation strategy drops the historical sharing of the kernel-level virtual-to-physical translation metadata (namely, the kernel-level page tables). In user mode, a process only observes a minimal amount of data and code belonging to the kernel, i.e., the data and code, to allow a safe transition to kernel mode upon interrupts and system calls. This minimal set of code, when activated, performs a page-table switch, which allows accessing the whole virtual address space of the operating system kernel. Before returning to user space, the user-land trimmed page table is put back in place. All major operating systems have adopted this scheme.

The main problem with this approach is that, for it to work correctly, a page table change must be accompanied by a flush of all virtual-to-physical translation entries in the caches—some of these are done automatically when

updating the page-table pointer, e.g. `CR3` on x86 CPUs. This incurs additional runtime costs, which have been quantified to be up to 30% of the execution time observed when this mitigation is not in place [59].

Concerning Spectre attacks, CPU vendors have introduced hardware mitigations for Speculative Store Bypass (SSB) [69, 8]. Examples are the Indirect Branch Restricted Speculation (IBRS) mitigation, which restricts speculation of indirect branches, the Single Thread Indirect Branch Predictors (STIBP) mitigation, which prevents indirect branch predictions from being controlled by the sibling hyperthread, and the Indirect Branch Predictor Barrier (IBPB) mitigation, which ensures that earlier code's behavior does not influence later indirect branch predictions. The software community has been cold towards these mitigations, as they have been reported to slow down typical workloads up to 50% [35].

On the other hand, one software mitigation to Spectre-like attacks, which introduces a minimal overhead, is the retpoline [77]. It is a software construct that ensures that if the CPU is mispeculating due to some attack being carried out against some branch prediction unit, then the pipeline will be filled with an infinite loop. This prevents arbitrary code execution, which could also induce data leaks.

Our work grounds on these and other mitigation patches and strategies. However, our goal is to enable these patches at a very fine grain, i.e., whenever a process is suspected as malicious. This finer grain should reduce the performance impact in the general lifetime of the system while ensuring a higher security level with respect to an utterly unpatched system.

4.2 Threat Model

We consider an attacker trying to carry out a cache-based attack and extract information from a co-located victim on the same platform. The attacker is thus sharing some architectural components with the victim, such as the First-Level Cache (L1) [133, 82] or the Lowest-Level Cache (LLC) [57, 74, 93, 79]. In the most general setting, the victim can be some userspace process, a virtual machine in a multi-tenant cloud environment, or the underlying OS kernel. We do not make assumptions on the privileges with which the attacker is running, nor on whether the side channel is being used to extract leaked information *after* some transient execution attack has been carried out. Indeed, as mentioned, we are interested in detecting the usage of a cache side channel to extract information while the attack is in progress. This allows us to detect also popular attacks such as Meltdown [92], Spectre [84], and Foreshadow [148, 157].

Concerning transient attacks, we assume that security mitigation patches such as KPTI are not necessarily active (hence the attack is not prevented)

but are available in the compiled operating system binary. Indeed, we propose a mitigation mechanism that allows to selectively re-enable these patches on a per-process basis, just to prevent the attack, if a process is suspected as malicious, as we shall describe in Section 4.5.

We also assume that the operating system's kernel is not compromised in any way. In particular, we assume that any data acquired by the kernel is not tampered with by an attacker and that the routines executed by the kernel are similarly not altered by any attacker. Hence, in our proposal, we assume that no attack is run from kernel space, e.g. the victim has not loaded any malicious kernel module that would mount a side-channel attack. The operating system's kernel internal and external security is an orthogonal security aspect to the proposal discussed in this work.

4.3 Detecting Side-Channel Attacks

The overall architecture and methodology that we use to enable prompt detection of side-channel attacks are depicted in Figure 4.1. We rely on a combination of measures taken from HPCs in real-time, which allows us to discriminate processes that are more likely to perform operations on the cache hierarchy, indicating that they are mounting a side-channel attack. At the kernel level, we have four major components involved in the system-wide monitoring of the attacks to detect the activity of malicious processes. The *Monitor* module directly interacts with hardware performance counters, programming them to acquire the measures to build our detection metrics. Data coming from HPCs are stored directly in a process' `task_struct`. The *Detector* module relies on these data to compute detection metrics and deem a running process as suspected or not—again, this information is stored in the task struct. If a process is suspected, the *Mitigator* module will detect it and apply proper mitigations. The *Scheduler* module interacts with the operating system's scheduler. It is one of the fundamental components to enable system-wide detection and per-process mitigations: every time that a different `task_struct` is scheduled, both the *Mitigator* and the *Monitor* modules are notified to enable/disable mitigations and reprogram HPCs, respectively, to account for the newly scheduled process.

4.3.1 Architectural Details and Preliminary Work Hypotheses

Since we are interested in the malicious usage of caches to extract information, it is beneficial to discuss how the CPU interacts with the caching subsystem briefly. Its organization is depicted in Figure 4.1 for the Intel architecture, which we use as our reference, where there are three levels of

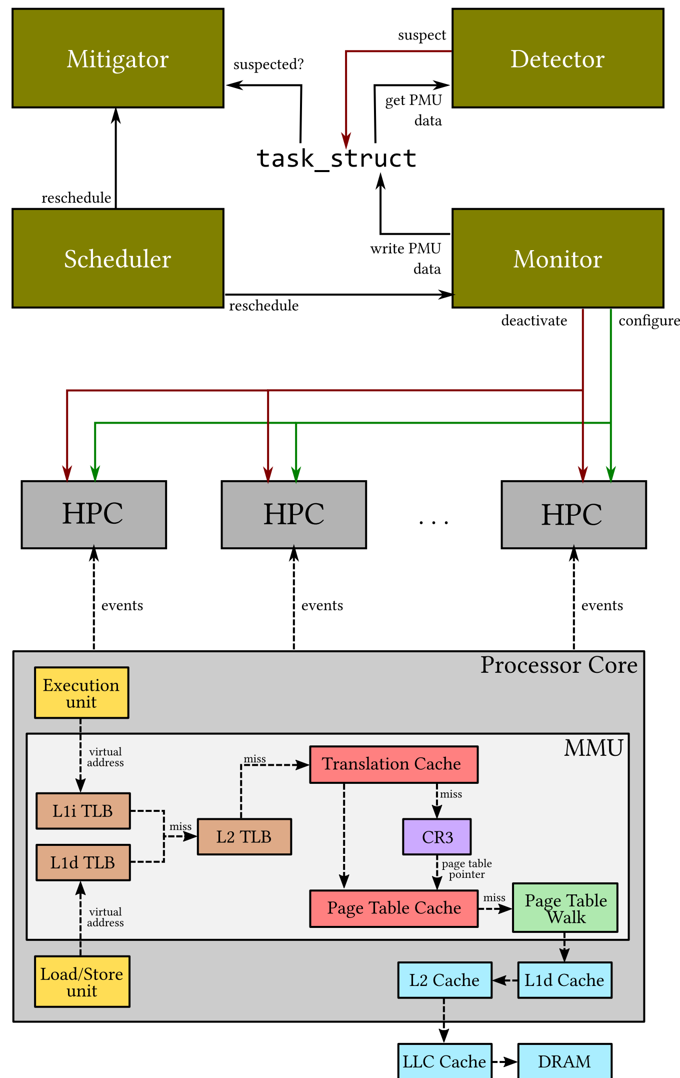


FIGURE 4.1: *Overall detection architecture and components involved in memory access—hit paths are not shown.*

CPU caches. The caches closer to the CPU are smaller and faster, and the caches further away are larger and slower. At the first level, there are two caches, L1i and L1d, which keep code and data, respectively. The L2 cache unifies code and data and, in almost every x86 CPU, represents the last cache level private to the core, while, as the final level, there is the Last Level Cache (LLC), a shared memory level among all cores of the same die. Two important cache properties to be considered are *inclusiveness* and *associativity*. The former defines the way a cache level behaves with respect to the higher ones, which can be:

1. inclusive: this level always contains data stored in the higher levels;
2. exclusive: precisely the opposite of the previous one;
3. non-inclusive: it does not guarantee that higher levels state is a subset of the current one.

This work considers inclusive caching systems since they represent the most diffused chipset for Intel processors. On the other hand, associativity is a strategy that divides a single cache level into multiple sets, where part of the physical address is used to index into the corresponding cache set. It is helpful to reduce chip complexity while providing a more efficient cache implementation. Also, the page table walk firmware relies on the CPU caches to further improve the performance of a TLB miss [149]. In particular, the PTEs for the different page table levels are not stored only in the CPU caches, but modern processors also store them in page table caches or translation caches [15]. Independently of the associativity strategy, we work at the granularity of the single cache line.

4.3.2 Detection Metrics

In this Section, we describe the metrics we have devised to determine whether some malicious side-channel activity is going on. In an initial set of metrics, we relied on the idea that the exploitation of a side channel is based on bringing the caching system into a known initial state. Successively, the attacker attempts to determine whether some change has occurred in the cache state. However, considering inclusive caching systems, which represent our target, we know that bringing the cache into a given state (e.g., a cache line is flushed or a cache set is primed) means performing an operation that is necessarily reflected into the state of caches at all the levels, from L1 to LLC.

Based on this observation, we decided to relate to each other volumes of micro-architectural events that are generated at different levels within the caching system. At the same time, we wanted to focus on events that are

not easily manipulable (in terms of their volume generation at a specific cache level) by an attacker[2]. Therefore, we decided to avoid considering cache hits and to focus exclusively on cache miss events.

Independently of whether the initial state of the cache when the attack is started is based on cache-line flushes or cache-set primes, a side channel is anyhow based on re-accessing the same cache line to discover changes into the state. Clearly, the "interface" for the observation is the L1 cache, but the actual cache access needs to pass through lower levels if a miss is observed. On the other hand, if a miss has been experienced at the L1, the likelihood of observing misses at the lower levels is expected to be high. In fact, by the nature of a cache side channel, the victim either brought some cache line into the caching system, up to L1, or gave rise to cache line replacement involving all the cache levels because of inclusiveness.

Considering that in inclusive caching systems the volume of cache misses at upper levels is greater than or equal to the ones at the lower levels, our first two detection metrics are based on the ratio between the number of cache misses at L1 denoted as $L1_{miss}$, and the corresponding values at lower levels denoted as $L2_{miss}$ and LLC_{miss}. We have therefore two predicates $\mathcal{P}_1$ and $\mathcal{P}_2$ for building our side channel suspicion, which are based on relating the aforementioned ratios to thresholds, namely:

$$\mathcal{P}_1 : \quad L2_{miss}/L1_{miss} > \phi_1 \tag{4.1}$$

$$\mathcal{P}_2 : \quad LLC_{miss}/L1_{miss} > \phi_2 \tag{4.2}$$

with the values of ϕ_1 and ϕ_2 both included in the interval [0,1]. Clearly, the value zero for these thresholds leads to a highly conservative setting where the predicates always hold, leading to suspicion independently of the actual execution pattern. Values closer to one are more representative in terms of the ability to discriminate between malware and benignware.

Another interesting point about caching is that the caching hierarchy typically supports data prefetch in order to implement anticipated reads useful to serve access locality by the applications. For example, this is the case of the L2 cache in Intel processors. However, a cache side channel is typically based on activities that target a specific cache line. Hence, we may expect that prefetched data may result useless. On the other hand, the scarce exploitation of prefetched data is challenging to be discovered by relying on miss events. For this reason, we devised an additional metric based on the relation between the number of write-back operations for cached lines at the L2 cache, which we denote a $L2_{write-back}$ and the number of lines fetched (including the prefetched ones) still at the L2, which we de-

[2] As an example, an attacker might easily give rise to volumes of cache hits at the L1 in an uncorrelated manner to the cache hits observable at the LLC.

note as $L2_{lines-in}$. Accordingly, we derive a third predicate $\mathcal{P}_3$, based on an additional threshold ϕ_3, still having a value in the interval [0,1], in order to determine the side channel suspicion, according to the following expression:

$$\mathcal{P}_3 : \quad L2_{write-back}/L2_{lines-in} < \phi_3 \tag{4.3}$$

Essentially, predicate $\mathcal{P}_3$ is intended to capture all the scenarios where data update activities do not comply with locality expectations (especially for very low values of ϕ_3), which can be an indication of some unexpected non-local behavior.

The above-described metrics and predicates are tailored at direct cache side-channel attacks, namely those attacks that are based on managing cache lines/sets explicitly with data in the address space of the attacker. Another way of attacking the cache to mount a side channel is to have indirect attacks based on the fact that memory management metadata, in particular, page table entries, are still cached (see Figure 4.1). This may lead to evict cache sets with these metadata, thus enabling the determination of the metadata re-access time to discover whether some victims had conflicting accesses to the same cache line used to keep the page table entries. To cope with these kinds of attacks, we devised an additional metric, based on the number of TLB misses at the second level of the page walk, denoted as $TLB_{miss-level-2}$ and the number of L1 cache misses. In particular, a high value of the ratio between TLB misses and L1 misses is representative of a behavior not conforming with classical locality (namely, a behavior not conforming with good exploitation of already carried out virtual-to-physical address translations). This may therefore be a behavior where a cache miss is generated just because of the will to fill the TLB (upon a TLB miss) with data leading to a cache line replacement. In order to determine the side channel suspicion in such indirect attack scenarios, we have therefore the following additional predicate:

$$\mathcal{P}_4 : \quad TLB_{miss-level-2}/L1_{miss} > \phi_4 \tag{4.4}$$

where ϕ_4 is this time not constrained in the interval [0,1].

At this point, we can combine the above-defined predicates to determine whether to suspect that a side channel exploitation is taking place, or not. Before doing this, for direct side-channel attacks, we exploit again the ratio $TLB_{miss-level-2}/L1_{miss}$ to define the following additional predicate:

$$\mathcal{P}_5 : \quad TLB_{miss-level-2}/L1_{miss} < \phi_5 \tag{4.5}$$

with $\phi_5 < \phi_4$. Actually, $\mathcal{P}_5$ expresses the fact that there is no bias generated by a direct side-channel attack in terms of the increase in the volume of

TLB misses, with respect to the volume of L1 misses related to actually-accessed data. In fact, direct attacks only exploit data in the address space (not memory address translation metadata). Overall, for direct attacks we define a combined predicate S_1 to determine whether to raise a suspect as the following combination of $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_5$:

$$\mathcal{S}_1 = \mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \mathcal{P}_3 \wedge \mathcal{P}_5 \tag{4.6}$$

For indirect attacks, we just have $\mathcal{P}_4$. Finally, a side channel suspicion is raised based on the following combination of $\mathcal{S}_1$ and $\mathcal{P}_4$:

$$\mathcal{S} : \quad \mathcal{S}_1 \vee \mathcal{P}_4 \tag{4.7}$$

4.3.3 Setting up the thresholds

Basing the detection on measures from HPCs compared against thresholds has already been identified in the literature as a possible pitfall [38]. However, a significant identified issue is related to how these thresholds are set and how they are employed. In particular, while it is clear that thresholds could be circumvented (e.g., by inducing page faults to affect the accuracy of the measured events [38]), we emphasize that we play on the safe side. Indeed, we use thresholds to discriminate between *relations* among events, which are in any case representative of the ultimate utilization of the side channel to extract information. Furthermore, our approach to side channel detection is based on combining metrics (via the combination of predicates involving these metrics), which should favor robustness.

Nevertheless, relying on hardcoded thresholds would make the approach difficult to maintain over time, requiring significant manual intervention. Changes in the hardware, or peculiarities of specific CPUs, are some of the aspects which could require to re-tune the thresholds.

The behaviors which we discriminate with our metrics depend mainly on the architecture of the cache of the machine in which the thresholds are used to discriminate a process as malicious or not. To this end, our system explicitly allows defining the values for the different thresholds ϕ_i for the actual machine on which we perform the detection at configuration time. In our reference implementation—see Section 4.6 for additional details—this is done by running:

- A set of side-channel attacks in a controlled environment.
- A set of benchmark applications from different fields.

By relying on these attacks and on the behavior of the benchmarks (which represent the benignware part), we can estimate proper values for

the thresholds, which are used in our detection mechanism. In particular, we define a threshold value as the average of the two (already averaged values) for the cases of the run attacks and benchmarks. We note that including benignware execution in the setup of the thresholds gives rise to a somehow conservative estimation of the threshold values that is, anyhow, not unfavorable to non-malicious software.

We note that to avoid bias in the experiments, the synthetic attacks which we carry out at system startup are different from the ones which have been used to test our approach. This is an approach similar in spirit to techniques that perform preliminary probing of the hardware architecture in order to carry out an attack effectively [150].

4.4 System-Wide Detection and Reference Implementation

As mentioned, our goal is to carry out a system-wide detection of possible attacks relying on side channels to extract information. This detection is carried out at the kernel level—our reference implementation is based on a set of patches applied to Linux 5.4.145. In our implementation, we have targeted the Intel architecture, considering its widespread nature [121] and the fact that it has been repeatedly subject to multiple attacks in the last years. Nevertheless, as we discuss, our reference implementation can be easily ported to other architectures, such as AMD.

4.4.1 Selected Monitoring Events and Strategy to Acquire HPC Data

We must first give additional details on how we have configured HPCs. Sampling has been set to follow the number of clock cycles—`CPU_CLK_UNHALTED` on Intel CPUs, `PMCx076` (*CPU Clocks not Halted*) on AMD. With respect to the measures, we have tried to select stable measurements to instantiate the proposed metrics. In particular, the following events have been selected [67, 9]:

- $L1_{miss}$ is mapped to the `L2_RQSTS.ALL_DEMAND_DATA_RD` event. On AMD, a suitably corresponding event is `PMCx041` (*Data Cache Misses*);
- $L2_{miss}$ is mapped to the `L2_RQSTS.DEMAND_DATA_RD_MISS` event. On AMD, a suitably corresponding event is `PMCx07E` (*L2 Cache Misses*);
- LLC_{miss} is mapped to the `OFFCORE_RQSTS.L3_MISS_DEMAND_DATA_RD` event. On AMD, a suitably corresponding event is `PMCx0E0` (*DRAM Accesses*);

- $L2_{write-back}$ is mapped to the `L2_TRANS.L2_WB` event. On AMD, a suitably corresponding event is `PMCx07F` (*PMCx07F L2 Fill/Writeback (L2Writebacks bit set)*);
- $L2_{lines-in}$ is mapped to the `L2_LINES_IN.ALL` event. On AMD, a suitably corresponding event is `PMCx07F` (*PMCx07F L2 Fill/Writeback (L2Fills bit set)*);
- TLB_{miss_level2} is mapped to the `DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK` event. On AMD, a suitably corresponding event is `PMCx045` (*PMCx046 Unified TLB Miss*).

We have configured our implementation for scenarios where Simultaneous Multi-Threading (SMT) is disabled. This choice is motivated by the fact that, with SMT disabled, Intel CPUs of various generations offer at least eight programmable HPCs, which are enough to sample all the parameters involved in our metrics. With SMT enabled, this number would be reduced to four on many CPUs. Furthermore, by disabling SMT, we remove the noise in the experimental assessment related to the need to time-share the HPC units to gather data related to different measures, thus focusing better on the validity of our approach. Using our approach with SMT enabled is possible, but it requires techniques to share HPCs to read multiple measures, which are out of the scope of this work.

In our reference implementation, we have tackled the cost of running the PMI handler by installing a custom interrupt handler lined up on a free vector in the Linux Interrupt Descriptor Table (IDT). This custom interrupt handler, which is reserved for PMIs, bypasses the traditional activation scheme for interrupt management in Linux. Indeed, Linux typically manages a hard interrupt by activating multiple nested functions, in particular related to the identification of the proper Interrupt Service Routine in charge of managing the IRQ. This is a cost that cannot be paid to just record a number of occurred events from an HPC.

Our custom stub accounts for the bare minimum amount of actions required to serve the interrupt request (namely: possibly execute `swapgs`, change the page table if KPTI is active, set the per-CPU flags used to determine the execution in kernel mode, take a CPU snapshot). After the actual mode change, we filter out possible spurious interrupts, and we collect samples from HPCs. We then compute our detection metrics[3], determine whether to consider the current process as suspected or not, and finally return from interrupt. The obtained data are saved on a per-process basis

[3]We have implemented metrics evaluation in integer arithmetic, both to reduce the execution time and not to poison the FPU—we are not explicitly saving the FPU state.

in the `task_struct` of the thread currently running on the core, which is serving the interrupt request.

Another important aspect is that HPCs are shared among processes scheduled on it. As shown in [38], underestimating this property leads to an inconsistent behavior of the system-wide detection mechanism. To cope with this aspect, upon context switch (`prepare_task_switch()`), if the observed data is enough, we early evaluate the metrics for the about-to-be-descheduled process. On the other hand, before returning control to the newly-scheduled process (namely, in `finish_task_switch()`), we logically reset the HPCs and start the measurement of the about-to-be-scheduled process. In this way, we do not mix HPC data coming from the execution of different processes if a thread is scheduled/descheduled in the middle of an observation window, which could lead to an erroneous detection.

As a last note, we have configured HPCs to explicitly filter out activities when running in kernel mode—the `USR` configuration bit in the HPC control register. In this way, every time that we run in kernel mode (also to extract the values of some HPCs upon a PMI), we do not overcount the measures taken from HPCs—this solves another source of unreliability observed in [38][4].

4.4.2 Observation Windows

As discussed, an effective system-wide detection requires filtering out all the activities not directly related to the instruction sequence of the attack to avoid pollution in the observed data. Given that HPCs cannot leave the micro-architectural domain, it is impossible to identify program phases just by counting low-level events. This aspect could allow an attacker to blend the malicious code into any program, concentrating the attack phase to a limited execution time window.

To cope with this problem, we divide the entire observation period into *time slots*, which are handled as observation windows of HPC values that are inspected one by one. This allows discriminating among different execution phases. Such a discretization is applied to the number of *elapsed clock cycles* (which defines a constant unit among all running processes) rather than events such as *retired instructions*—they may warp the time slot depend-

[4]Unfortunately, Intel confirmed [71] that using CPL through the PMC's `OS-USR` bits may lead to an incorrect result, such that the sum of OS-data and USR-data is not equal to the result obtained by counting without filters. This phenomenon is probably due to the high out-of-order execution degree, which makes it hard to associate the μop execution with the correct execution ring near a mode transition. Nevertheless, for our specific context, we have performed some tests in order to quantify this error and observed very minimal error values (less than 0.1%).

ing on the executed instruction[5]. This window is preserved across context switches and is not shared among processes/threads, thus guaranteeing a coherent inspection of the execution flow. In other words, if a process/ thread is descheduled in the middle of an observation window, once it is rescheduled, we resume collecting HPC data from the same exact "point" in the observation window at which it was descheduled. We also note that this approach allows overcoming the problem affecting other works (see, e.g., [110]), in which data collection is associated with the entire program execution.

The related HPC's overflow defines the beginning and end of a time window. It is essential to determine the time slot size so that the observable data is enough to discriminate meaningful program phases. A too-small size may cause each slot to provide noisy and poor information, while a too-large one will eventually fall into the same pitfall as in [110], i.e., too much-aggregated data. Furthermore, the size of the time window is directly related to the overhead that the detection architecture introduces in the system because smaller slots imply more interrupts to be processed.

Similarly to what we have discussed in Section 4.3.3, we determine the minimum and maximum thresholds for the observation window at system startup, guaranteeing stable measurements. This is done via an adaptive approach: if we observe a large fluctuation in the data observed across two consecutive windows, we reduce the size of the window (up to a compile-time defined minimum threshold, which accounts for the overhead in the measurement). Conversely, if variations are minimal, we increase its size (up to another compile-time defined maximum).

4.4.3 Suspecting Malicious Processes

After calculating the metrics in the PMI, they are compared to the respective thresholds, thus determining if the predicates driving suspicion hold—see expression (4.7). Based on the inequalities results, we deem a process as malicious or not. Obviously, the classification of a process cannot be made based on a single observation because we would have an excessive number of false positives considering that, during its execution, a process can assume different behaviors. For this reason, we have introduced a scoring system. The process's score will vary during execution as follows:

- the score is increased by α if the results of the comparison between metrics and thresholds show a behavior similar to a side-channel attack;

[5] Every instruction requires a certain number of clock cycles to be carried out, which varies according to several factors (e.g., the memory state).

- the score is decremented by β if the metrics do not detect any abnormal situation.

If the score reaches the value of a threshold γ, then the process becomes suspected. α, β, and γ are tunable hyperparameters of our model. These parameters are related to each other in the following way. α indicates how fast a process becomes suspected: the higher the value, the smaller is the number of positive evaluations of the metrics required to flag it as malicious. Conversely, β determines how fast a process that was (incorrectly) considered suspicious starts again to be deemed benign. α and β can be therefore used to control the responsiveness of our scoring system towards punctual activities (i.e., possibly malicious or not) exhibited within an observation window. In the general case, we assume $\alpha \geq \beta$ to allow for a prompt-enough detection of a malicious process. Conversely, γ directly controls when a process becomes flagged as malicious. To some extent, it indicates the amount of data that the system tolerates to leak before deeming a process as suspected. In Section 4.6, we provide an empirical assessment of the behavior of our approach with respect to these parameters.

Once a process is suspected, this information is stored by exploiting bit 27 of `current->mm->flags`[6]. We have explicitly decided to rely on the `flags` field in the `mm` data structure because, upon a `fork()`, this data structure is automatically copied by the kernel, to make it inherited by the child process. In this way, also if the attacker tries to jeopardize our detection system by relying on a multi-process attack, the behavioral information associated with children and the parent processes is shared.

4.5 Mitigation Strategies

Our kernel-based detection subsystem can flag a process as suspected. A suspected process is one for which we can implement mitigations. We note that this is not a destructive operation: even if we have incurred a classification error (i.e., a false positive), the fact that we enable mitigations will not cause runtime errors (e.g., abnormal termination) in the wrongly-suspected process. Indeed, we could only cause a performance slowdown. Nevertheless, considering the overall system, this slowdown will not be comparable to that observed if the mitigations we discuss here were activated by default for all processes—see Section 4.6 for the overhead assessment. We have foreseen two families of mitigations: one related to side-channel attacks in general and one pertaining to transient execution vulnerabilities. The mitigations we put in place have value independently of whether our

[6]Bit 27 is currently unused.

approach is used to detect the attacks or other support would be used to determine (potentially) malicious processes.

4.5.1 Side-channel Attack Mitigations

Multiple mitigations belong to this family. The first one entails that, in `finish_task_switch()`, before returning control to a thread of a suspected process, we flush the last-level CPU cache. This ensures that no data from other processes is available in the cache to be leaked. Of course, this is an intrusive operation performance-wise. However, it mainly affects the execution of the suspected process, for which the cache must be again warmed up upon its reschedule[7].

A second mitigation we devised tries to mitigate the fact that an attacker is likely running on a CPU core that is "close" to the cache used by the victim. Therefore, an additional strategy is to change the affinity of the attacker to move it to a different core which is not sharing the same level of cache with the victim. We note that this mitigation could also be performance-intrusive, particularly for applications that have explicitly set their affinity, e.g., to control their memory-access latency on NUMA machines. However, the system administrator always can change the affinity for any thread. Hence our approach mimics such a kind of housekeeping job, in this case carried out for security purposes.

4.5.2 Transient Execution Mitigations

Another mitigation that we explicitly put in place is per-process enabling of KPTI. The baseline implementation of KPTI in Linux has been slightly changed to support this mitigation.

In particular, while we maintain the order-1 allocation (8 KB) for the first-level page table (pointed to by the CR3 register), which allows having two different views of the address spaces for each process (one for user mode, one for kernel mode), by default all processes rely on only one order-0 page table, which maps the whole kernel address space. This configuration resembles the traditional organization of the memory map in Linux before the introduction of KPTI. The two first-level page tables are kept synchronized following a scheme that resembles the one currently adopted to synchronize them whenever a userspace application allocates new physical memory. In particular, every time that a new set of physical pages is allocated, the

[7]We think that the performance penalty paid by the OS kernel when managing interrupts that do not find cached data after the reschedule of the suspected process can have a limited impact, with respect to the fact that upon the reschedule the CPU-core is anyhow devoted to the specific activities related to the execution flow of the suspected process.

kernel with KPTI enabled invokes `__pti_set_user_pgtbl()` which materializes in the user-level page table the newly-allocated virtual-to-physical translation metadata (the page table chain), also explicitly setting the `NX` bit, if available on the current architecture. This same scheme is adopted upon a `fork()`. We retain this scheme, although we explicitly differentiate between the user- and the kernel-level page table—this distinction is somewhat implicit in the current standard implementation of KPTI and relies on some hardcoded macros.

Upon a mode switch, the `switch_to_kernel_CR3` macro is used by the kernel to open access to the whole address space of the kernel. Upon return to userspace, the `switch_to_user_CR3` macro returns to the user-level page table. This scheme is done every time the machine transitions from user to kernel mode and vice-versa. Our goal is to selectively activate this scheme in a per-process way, reducing at most the cost for this operation.

To this end, we recall that a process becomes suspected while running in kernel mode, namely while a PMI is being processed. In that case, we set a flag in `current->mm->flags`. When returning to user mode, we explicitly check this flag. If it is set, we invoke `switch_to_user_CR3`. This is enough to start applying the patches for a suspected process. Conversely, we cannot check this flag when transitioning from user to kernel mode. This is because we do not have access to `current`, which is stored in per-CPU variables, which are not accessible if the user-mode page table is set.

To check if we have to invoke `switch_to_kernel_CR3`, we exploit the fact that the two first-level page tables belong to an order-1 allocation and are therefore contiguous (both in the virtual and in the physical address space). We have inverted the user and the kernel page table with respect to the current implementation of KPTI. This means that the user page table follows the kernel page table. Given the contiguousness of the pages, it is sufficient to check if bit 13 of the address contained in CR3 is set to 1. If this is the case, the thread enters kernel mode with the user-mode page table. This means that the thread belongs to a suspected process, and we, therefore, have to invoke `switch_to_kernel_CR3`. On the other hand, if the bit is cleared, the process is not suspected, and the whole kernel virtual address space is already visible.

Of course, we want to account for suspected multi-threaded applications explicitly. In this scenario, two threads could be concurrently running on multiple CPUs. We want to minimize the time window when a thread is running with patches enabled, and another is not. As mentioned, to activate the patch, a thread belonging to a suspected process must perform a mode switch from kernel mode. To this end, after flagging a process as suspected, we explicitly send an Inter Processor Interrupt (IPI) to all other cores. This operation will require all CPU cores to transition to kernel mode. In this

way, if a thread of the suspected process was running, the mode change will result in patch enabling.

A similar mechanism has been put in place to enable/disable several other mitigation techniques, namely: i) Microarchitectural Data Sampling (MDS); ii) Spectre v1, v2, L1TF mitigations; iii) SSB mitigations; iv) KVM Non-Executable Huge Pages; v) TSX Asynchronous Abort. This is supported by quickly checking the flag in `current->mm->flags` to determine whether one specific mitigation should be activated, which might, in turn, require modifying the content of some MSR value (as in the case of SSB mitigations).

4.6 Experimental Assessment

4.6.1 Experimental Setup

We have carried out an experimental assessment relying on multiple generations of Intel CPUs, namely using the following processors:

- i7-6700HQ 4x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 6MB 12-way;
- i7-7600U 2x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 4MB 16-way (with TSX);
- i5-8250U 4x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 6MB 12-way;
- i7-9750H 6x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 16MB 16-way;
- i7-10750H 6x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 12MB 16-way.

To set up the thresholds and observation windows used by our detection system, we have run versions of the attacks listed in Table 4.1, as well as the following set of benignware applications: (1) Firefox, with both textual page, multimedia content access, and browser benchmarks such as JetStream2; (2) VLC, with both large and short videos and random skip of video portions, as well as repositioning; (3) Evince Reader, with both small and large size pdf files, and random skip of pages; (4) gedit for editing

[4] `https://github.com/vusec/revanc`.
[5] `https://github.com/vusec/xlate`.
[6] `https://github.com/paboldin/meltdown-exploit`.
[7] `https://github.com/Eugnis/spectre-attack`.

textual files of different sizes and random positioning onto the file portion to be edited; (5) all the kernel-level threads operating within the Linux kernel.

TABLE 4.1: *An overview of considered cache side-channel attacks and references to the used implementations.*

Name	Same-Core	Cross-Core	Shared Memory	Measurement
EVICT + TIME [89] (taken from [8])	✓	✓	✗	time
PRIME + PROBE [74, 93] (taken from [9])	✓	✓	✗	time
PRIME + ABORT [44] (taken from [9])	✗	✓	✗	TSX
FLUSH + RELOAD [161] (taken from [9])	✓	✓	✓	time
FLUSH + FLUSH [56] (taken from [9])	✓	✓	✓	time
XLATE + TIME [149] (taken from [9])	✓	✓	✓	time
XLATE + PROBE [149] (taken from [9])	✓	✓	✓	time
XLATE + ABORT [149] (taken from [9])	✓	✓	✓	TSX
Meltdown (taken from [10])				
Spectre (taken from [11])				
Foreshadow [148, 157] (taken from [25])				

4.6.2 Stability of HPC Events

We evaluated HPCs stability in terms of both over-counting and determinism by comparing the data collected from HPCs with data obtained from software instrumentation—results are reported in Table 4.2. For this experiment, we relied on a basic (single thread) benchmark[12] which computes the first x prime numbers, where x is a user-defined parameter. As a baseline, we used cachegrind [112], which automatically detects the underlying cache structure and builds an equivalent cache model while executing the program. With cachegrind, we can compare the results related to memory accesses and cache misses—in this case, we are also able to assess, to some extent, the accuracy of HPCs. Nevertheless, L3 cache misses, L2 filled lines (it counts opportunistic events at cache line grain and includes prefetcher activity), and TLB miss (we use a specific event that requires the emulation of a second-level TLB) are not available. For these events, we compared

[12] `sysbench -test=cpu -cpu-max-prime=20000.`

TABLE 4.2: *Comparison between HPCs and Software Instrumentation on all the architectures. Err represents the distance (%) between HPCs and SW while HPCvar shows the HPCs variation coefficient.*

	i7-6700HQ				**i7-7600U**			
	HPCs	SW	Err	HPCvar	HPCs	SW	Err	HPCvar
loads	4494K	4744K	5.2%	∼0%	4494K	4744K	5.2%	∼0%
$L1_{miss}$	400K	308K	29%	2.0%	515K	308K	66%	2.9%
$L3_{miss}$	8557	-	-	6.4%	7798	-	-	3.9%
$L2_{lines}$	185K	-	-	7.9%	221K	-	-	3.1%
TLB_{miss}	9168	-	-	∼0%	9382	-	-	5.7%
	i5-8250U				**i7-9750H**			
	HPCs	SW	Err	HPCvar	HPCs	SW	Err	HPCvar
loads	4494K	4744K	5.2%	∼0%	4494K	4744K	5.2%	∼0%
$L1_{miss}$	521K	308K	69%	1.6%	513K	312K	64%	2.2%
$L3_{miss}$	6539	-	-	3.6%	8064	-	-	4.0%
$L2_{lines}$	224K	-	-	4.7%	227K	-	-	2.9%
TLB_{miss}	8981	-	-	1.9%	9321	-	-	2.4%
	i7-10750H							
	HPCs	SW	Err	HPCvar				
loads	4495K	4744K	5.5%	∼0%				
$L1_{miss}$	517K	308K	67%	1.8%				
$L3_{miss}$	3725	-	-	7.7%				
$L2_{lines}$	226K	-	-	5.5%				
TLB_{miss}	9301	-	-	3.0%				

the HPCs values of several runs to compute the determinism degree of this source. The results in Table 4.2 experimentally confirm that, although HPCs could be subject to reliability errors, we have selected events that are more stable and portable across different architectures. Although the L1 miss Err value may be a wake-up call to the reader, it is consistent among the tested architectures and the HPCs variation coefficient. This result stems from cachegrind's inability to model all the hardware counterpart's internal details that vendors do not disclose.

4.6.3 Accuracy of the System-Wide Detection Approach

To assess the capabilities of our detection system, we have performed a system-wide experimental evaluation by building sets of benignware and malware applications. The former relies on the Phoronix Test Suite [88], from which we selected 156 benchmarks (configured with different inputs) showing various behaviors and load profiles. Conversely, to build the set of

malicious applications to exercise our solution's capability to detect side-channel attacks, we have not found access to real-world malware of this kind. Consequently, we have crafted such malicious applications starting from the stress-ng suite [83]. We injected side-channel attacks (based on the implementations reported in Table 4.1) into various benchmarks of the suite, generating a set of 100 malicious applications. The side-channel routine is placed within the benchmark stress function[13]. The attack is anyhow enabled only after a random delay and, after its activation, the side-channel procedure executes with a specific probability—we set this probability to 10%. By introducing these sources of uncertainty, we increased the non-determinism degree that attacks may exploit in realistic scenarios.

As described in Section 4.4, the behavior of our detection system depends on the α, β, and γ hyperparameters. We set α and β to 1 for the entire experimental phase while varying γ to evaluate the detection according to different threshold levels. As discussed, α and β represent the rates that regulate the score progression of each process in the system. By setting $\alpha = \beta = 1$, we are identifying a critical scenario for our detection system, as we slow down the detection of malicious applications while reducing the possibility for a benign application to "recover" from spurious actions being detected as malicious. At the same time, by varying γ, we somewhat change the responsiveness to an undergoing attack. As stated in Section 4.3.2, we recall that $\mathcal{S}_1$ and $\mathcal{P}_4$ predicates identify, respectively, side channels directly exploiting the cache levels (L1, L2, LLC) and external caching structures (i.e., TLBs) to manipulate the processor caches indirectly. In our tests, indirect attacks refer to XLATE implementations.

Figure 4.2 shows the results of the detection accuracy as confusion matrices. The standard benchmarks (i.e., with no side-channel attack injected) are labeled as *OK*, while $\mathcal{S}_1$ and $\mathcal{P}_4$ indicate the direct and the indirect attacks, respectively. Confusion matrices with $\gamma = 1$ illustrate the behavior of our detection system as if the scoring system were not available. In this configuration, any application becomes suspected after a single violation of any metric. As we can observe, the number of false positives is non-minimal, and on the i7-6700HQ, it is even higher than real negatives. Overall, the benchmarks which have been wrongly suspected are the ones that either: i) involve a large number of `forks` and therefore propagate the information associated with the measures across a large number of processes; ii) implement data processing or machine learning algorithms iii) are memory-intensive scientific applications or explicitly test the memory hierarchy.

Nonetheless, the number of false positives quickly decreases as the value of γ increases. Indeed, this is related to the fact that subsequent obser-

[13]The stress function of each benchmark is called several times into a stress-ng main loop according to input parameters.

vations can filter out any potential spike in applications' activity without prematurely marking the process as suspected. This trend matches exactly our expectations, also validating the viability of the scoring system. Our experiments did not report any false-negative detection.

The approach we have proposed well fits scenarios in which a higher level of security is desired. However, the system is still prone to performance optimization under very low-security risks. Moreover, by design, the tuning mechanism aims to reduce the likelihood of experiencing false negatives at the cost of slightly increasing the number of false positives. Nevertheless, if γ is set to a suitably high value, this number becomes negligible.

By definition, a detection system is not a predictor, but it reacts to some events and makes decisions according to its model. Indeed, such a characteristic is crucial. Before classifying a malicious process as suspected, we expect part of its attack to have been executed—at least, the portion required to generate an identifiable pattern by our detection system. Typical side-channel attacks rely on a preliminary *preparation phase* (e.g., probing the cache) during which no data is actually read. If our detection system can detect a side-channel attack during this preparation phase, the attacker will not be able to read any data. Conversely, if the detection system identifies the attack during its *extraction phase*, then some amount of information might be read by the attacker.

Overall, the amount of data that an attacker can read even if our detection system is active is an important metric to assess the accuracy of our system-wide detection approach. Therefore, we have carried out an experiment to quantify the amount of data that a malicious process can read before its detection. In this experiment, the attacker shares a chunk of read-only memory with the victim and tries to leak information by mounting a side-channel attack on a byte-by-byte basis. Concurrently, the victim reads the shared buffer one byte at a time with some delay among subsequent accesses, generating all the conditions to perpetrate the cache-based attack. In this experiment, we have set the secret's size to 256 bytes—a non-minimal buffer corresponding to the size of a large Advanced Encryption Standard (AES) key—and studied the attack's effectiveness to extract data before being detected. Figure 4.3 shows the results of this experiment with detection capabilities turned on with different values of γ and different victim's read rates[14] . With $\gamma = 100$, our approach can detect the attack before it extracts a significant fraction of the data extracted when no detection was active, but only when the victim reads with small delays. By

[14]In this experiment, we have used an observation window of 2^{20}. A relation between the delay between two victim's reads (in μsec) and the observation window's size (in clock cycles) can be devised by considering that at a frequency of 1 GHz ($\sim 2^{30}$), we have, given the sampling period of 2^{20}, 2^{10} samples in one second. 1024 samples in one second correspond, roughly, to 1 sample/ms.

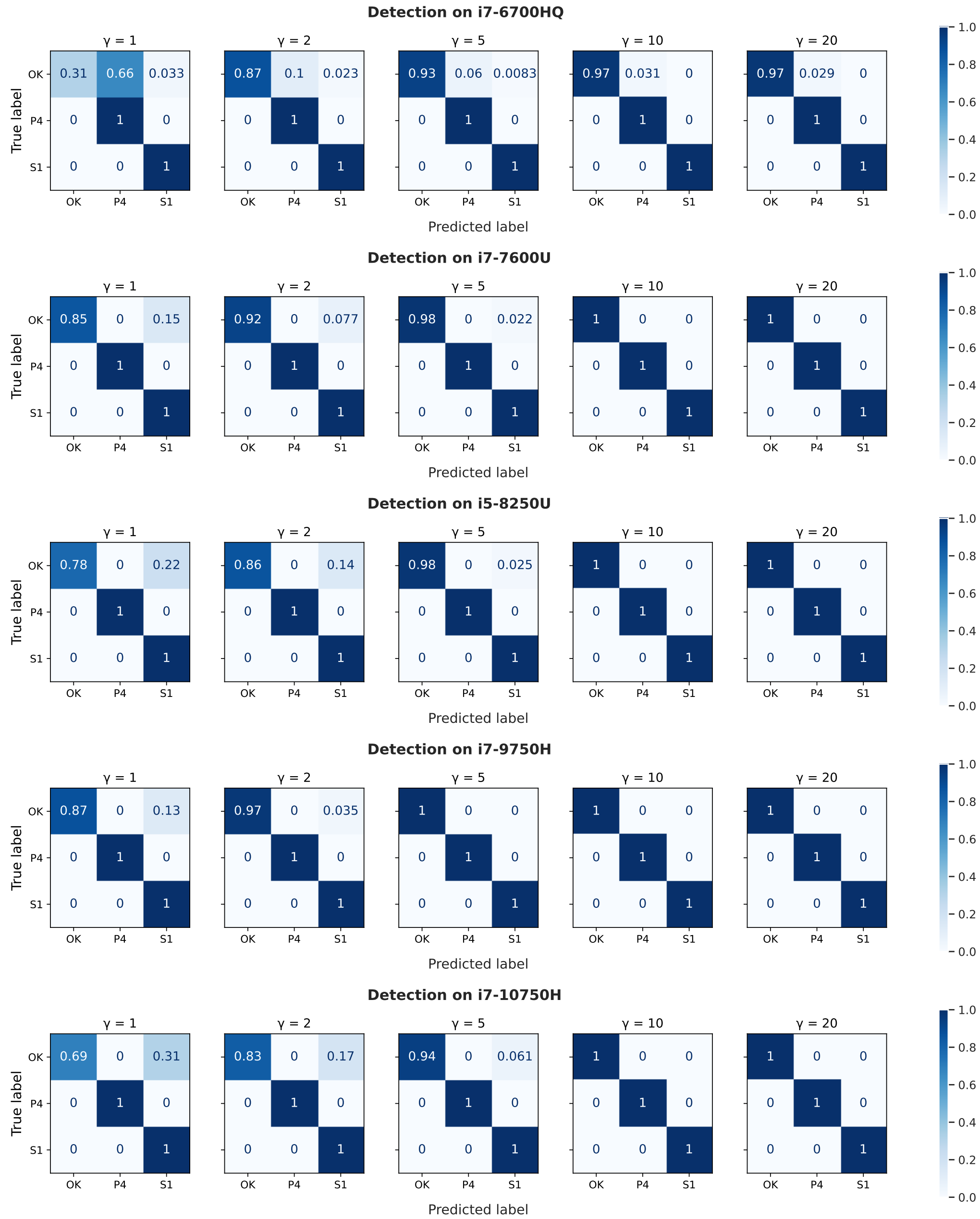


FIGURE 4.2: *Detection accuracy evaluation for different values of* γ $(\alpha, \beta = 1)$*.* $\mathcal{S}_1$ *and* $\mathcal{P}_4$ *indicate the direct and the indirect attacks, respectively, while* OK *indicates normal processes.*

decreasing γ, the percentage of correctly extracted data is reduced.

Although the reader may think that by increasing the victim's read rate, the detection may fail to identify the attack promptly, our results show that the percentage of extracted bytes decreases for very high-frequency reads on all examined architectures. This phenomenon is due to side-channel attacks being more sensitive to the noise generated when the activity in the system increases.

4.6.4 Performance Assessment

We have studied the performance improvement we can obtain with our monitoring proposal. To quantify the performance benefit of our approach, we have again relied on the Phoronix Test Suite, selecting a set of benchmarks that interact with the system in different ways, according to the following classes of behavior:

(A) intensive disk I/O operations (`compilebench`);

(B) pressure on the scheduler and context switch operation, also considering multithreaded applications (`hackbench`, `ctx_clock`);

(C) a large number of system call invocations, such as `fork`, `exec`, and those related to memory management (`OSBench`);

(D) high usage of the network socket API (`sockperf`);

(E) high usage of the GNU C Library APIs (`glibc-bench`);

(F) complex workloads, related to browsers and databases (`selenium`, `sqlite-speedtest`, `Apache`).

We also note that selecting these benchmarks allows profiling different classes of applications, namely CPU-bound ones (in userspace) or applications that repeatedly interact with the kernel, forcing the application to make a substantial number of mode switches. Given the implementation of our software patches, we should have an influence on the performance of the considered applications. No side-channel attack has been mounted in this experiment.

These benchmarks were run in the four following scenarios to evaluate the performance impact of the system-wide detection scheme, also accounting for the effect of the observation window's length:

(A) Mainline kernel 5.4.145 with KPTI, retpolines, SSB mitigations, and all the patches discussed in Section 4.5 enabled by default for all processes—referred to as `Generic` in the plots.

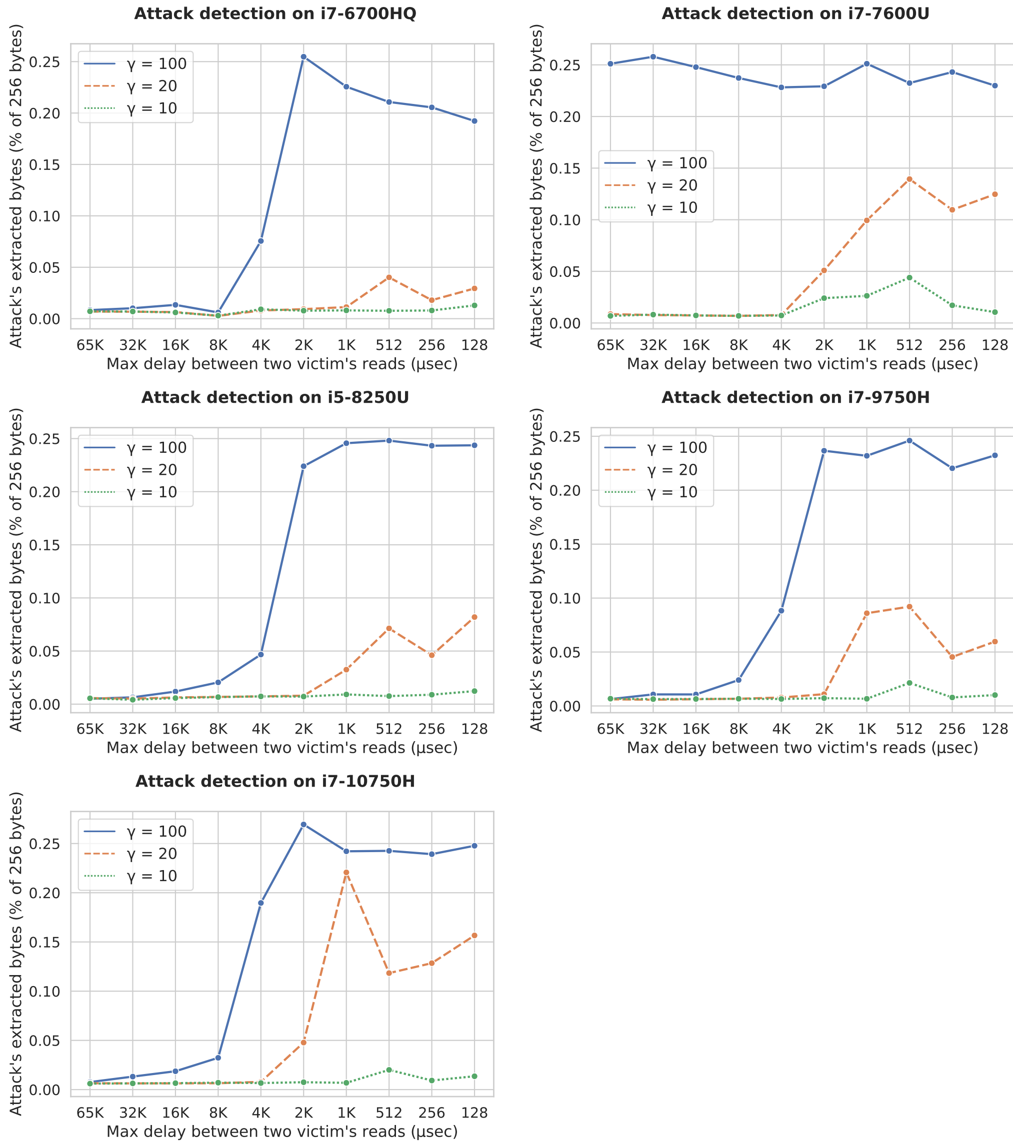


FIGURE 4.3: *Percentage of a 256-byte secret that an attack can correctly extract before its detection for different values of* γ $(\alpha, \beta = 1)$ *and victim's read rates.*

(B) Kernel 5.4.145, with our support for dynamic patching, but with system-wide monitoring disabled—referred to as `Monitor OFF` in the plots.

(C) Kernel 5.4.145, with our system-wide detection scheme activated, with an observation window set to 2^{20} clock cycles, which was the minimum observation window value considered by the adaptive approach—referred to as `Monitor (short window)` in the plots.

(D) Kernel 5.4.145, with our system-wide detection scheme activated, with an observation window set to 2^{24} clock cycles, which was the maximum observation window value considered by the adaptive approach—referred to as `Monitor (long window)` in the plots.

The results for the benchmarks in these configurations are reported in Figure 4.4, where we show the overhead with respect to the mainline kernel 5.4.145 with no active patch, which is therefore vulnerable to all the discussed attacks—values are averaged over three different runs. By the results, we can observe that the `Monitor OFF` approach offers a performance slowdown with respect to the `Generic` configuration, which is up to 4 orders of magnitude lower while showing an overhead over the unpatched mainline kernel lower than 4% on all architectures and for all application classes. This means that the support we have introduced in the kernel to enable/disable at runtime the various security patches is lightweight and non-intrusive.

Conversely, the overhead of the `Monitor` configuration over the `Monitor OFF` configuration is negligible.

It is interesting to note that the impact of the window length is minimal: considering that they are related to the maximum/minimum values supported by our system, this experiment shows that the expected overhead, also accounting for the adaptive optimization of the window, is reduced. Of course, this reduced overhead is coupled with our proposal's increased security level. Overall, this is additional evidence of the viability of our proposal.

A similar trend can be observed for all tested architectures (except i7-10750H) and all classes of applications, although with different relative ratios. This indicates the stability of our approach with respect to the performance of applications. The results on the i7-10750H processor do not match the other models' behavior. This is because Intel, starting from the 10th generation of its processors, introduced design changes to patch some hardware vulnerabilities. Consequently, the Linux kernel does not require enabling all the software patches (such as KPTI) on these processors with a mitigation of the performance slowdown. Nevertheless, our approach can still detect side-channel attacks on more modern architecture for which a hardware patch has not been proposed, with reduced overhead.

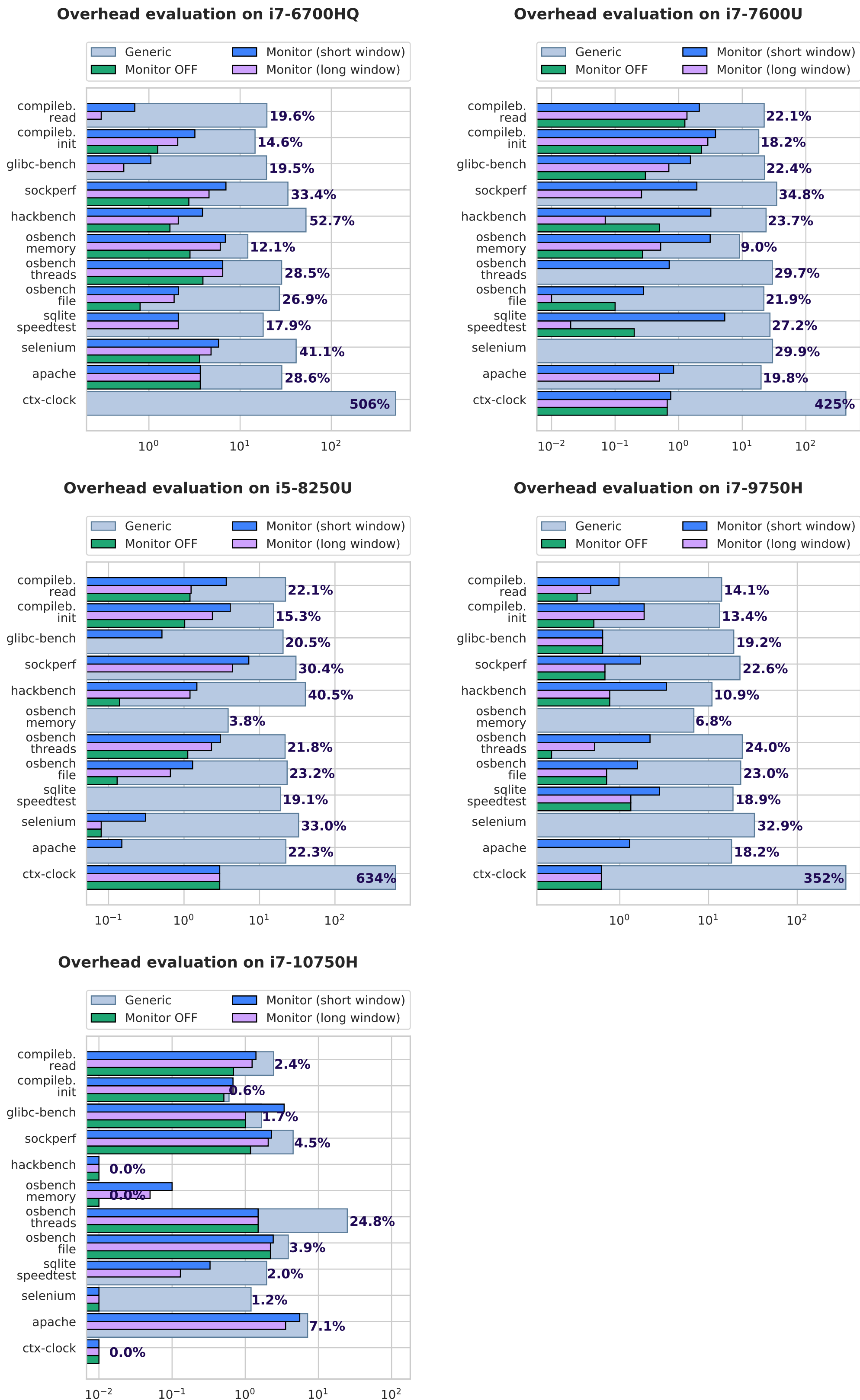


FIGURE 4.4: *Performance Effects of the HPC-based Monitoring System on different Architectures (logscale on the x axis).*

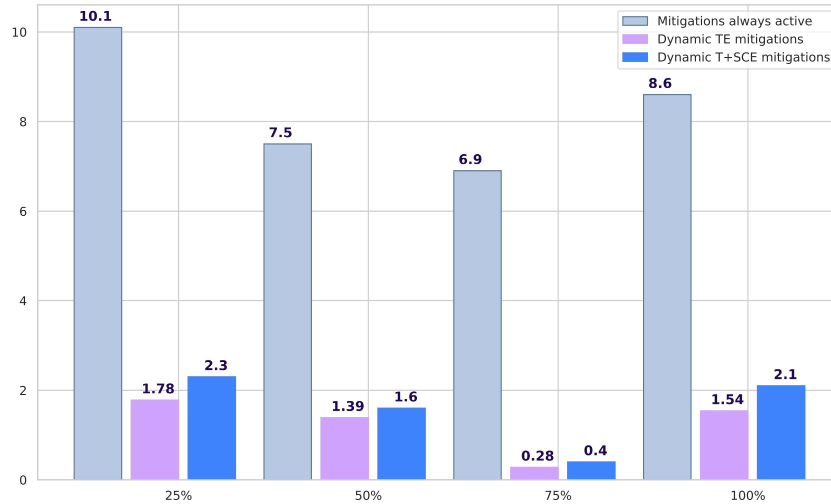


FIGURE 4.5: *Performance Penalties by Mitigations on the i5-8250U.*

The last experiment we present—the data are reported in Figure 4.5—relates to an assessment of the overhead due to transient execution mitigations and side-channel mitigations, also when there is significant interference with benignware on the same CPU cores. For this experiment, we only report data taken on the i5-8250U machine for the sake of space. In any case, the results on the other architectures show trends that are perfectly comparable with the data reported on this CPU.

We have launched a number of benchmarks taken from the Phoronix Test Suite equal to the number of available cores on the considered processor. Each benchmark has been statically pinned to one CPU core. We then varied the number of malicious applications, pinned to specific CPU cores, and ran them concurrently with the benignware benchmarks. This setup stress-tests also the per-process detection/mitigation capabilities of our system. We report data associated with the system run with all transient execution mitigations always active (`Mitigations always active` in the plot), with transient execution mitigations activated only for suspected processes (`Dynamic TE mitigations` in the plot), and with transient execution/side-channel mitigation countermeasures activated only for suspected processed (`Dynamic TE+SC mitigations` in the plot). The applications have been selected to avoid any false positive/negative. As in the previous experiment, we report the overhead as the percentage increase over an execution in which no mitigation at all (neither static nor dynamic) is present in the system.

By the result, we observe again that enforcing dynamic mitigations provides a significant overhead reduction, as high as 95%. As expected, the

overhead incurred when also `SC` mitigations are active is higher. Of course, depending on the system's configuration, the user can determine what set of mitigations should be enforced upon the detection of a malware application.

CHAPTER 5

Microarchitectural-driven application co-scheduling for system consolidation

Nowadays, one of the main challenges we are faced with is how to use computing infrastructures also making parts of the system (or the whole system) be effectively exploitable by multiple concurrent applications. This is one of the core objectives of *consolidation* [159, 61], which has emerged as the main topic in the area of virtualised systems and cloud computing but is a crucial target independently of the specific technology used for the actual deployment of software.

In this context, the co-location of applications or virtual machines on the same hardware platform can give rise to joint usage of the same components, like caches and buses, as well as devices. Looking at the co-location aspect from the point of view of HPCs, we observe that these concurrent applications can leave onto the hardware a given footprint depending on the specific features characterising the usage of the hardware components. As an example, a given co-existence can give rise to higher—or lower—amounts of misses per time unit at the different levels of the caching hierarchy depending on the features of the applications or even the workload they have been set to manage. Also, we may have a higher—or lower—rate of instruction commitment at the CPU-core level, just depending on how the current instruction flow(s) exploit reservation stations at the level of the CPU-core engine.

Clearly, HPCs can determine the applications' footprint on the hardware. However, an important step ahead when dealing with consolidation and sharing of resources can be represented by the exploitation of the hardware footprint to determine its trends and how these trends depend on the co-existence of the activities on the hardware. Knowing these trends, we can decide to make applications jointly use the hardware according to policies (and schedules) that are not currently supported by the operating system technology. This can represent an added value leading to the improvement

of the effectiveness of the hardware usage when hosting a specific set of applications.

In this chapter, we precisely study this problem. In particular, we work at the operating system kernel level, and we present a gang-scheduler of applications (particularly of applications' threads) hosted by the operating system. This scheduler is logically layered on top of the conventional CPU scheduler. It determines the wall-clock-time intervals where a specific subset of applications can share the hardware usage, thus working as a gang on the hardware. In contrast, other applications will use the hardware in a different wall-clock-time slice. The gang-scheduled set of threads is determined to give rise to better hardware footprints, which translates into higher effectiveness of hardware usage by the applications. With this solution, we target scenarios where long-running applications (like scientific or data management applications) need to massively use the hardware for a while, up to their turnaround. Still, we have no response time requirement related to external interactive entities—which may be non-compatible with the gang-scheduling approach.

In this solution, we start from a high-level hardware usage parameter, not directly linked to the outcomes we can get from HPCs, which is the CPU usage. This parameter allows us to determine what applications can still exploit the underlying hardware adequately—keeping the CPU not underutilised —when dispatched within the same gang. The gangs established by relying on the CPU-usage parameter are then analysed in terms of footprints on the hardware to determine what gangs are keeping the CPU used adequately and making individual components within the chip-set work at higher levels of their effectiveness.

Finally, after the analysis, the best gangs are actually selected for the software operations. It is important to note that this approach does not require any off-line analysis of the application behaviour before the applications are actually executed (possibly with particulars/new data as input). Instead, the gangs for the determination of the hardware footprints are identified at run-time and can also be updated by running the procedure periodically.

The original contributions of this chapter can be summed up as follows:

- We propose a new methodology that combines different high-level metrics and considers the software's activity from a hardware point of view to classify workloads;
- We describe a technique to explore different co-schedule at runtime without affecting the CPU quantum that the OS scheduler would assign to each of the involved processes;

- We present a modular infrastructure that manages registered applications and autonomously directs the OS according to the evaluation phase outcome.

5.1 Related Work

As pointed out, our objective is to include, at the operating system kernel level, a new layer that enables improvements in the footprint on the hardware by the applications via gang scheduling. However, the idea of improving hardware effectiveness through policies and mechanisms that drive the execution of software applications has been studied in the literature.

As for proposals that do not explicitly rely on hardware events and profiling data (like the ones coming from HPCs), we need to consider all the services or platforms that entail mechanisms for making any thread more likely able to exploit the memory hierarchy. For Non-Uniform-Memory-Access (NUMA) hardware, we have a baseline operating system support for defining data placement in the different NUMA nodes [46]. Furthermore, we also have higher-level solutions that exploit these operating system services to accommodate the placement (of both threads and data, through affinity mechanisms) along the execution of the applications. The approaches in, e.g., [58, 36, 42, 41] are general-purpose, hence being usable with applications from different domains. The solution in [123] is specific to high-performance simulations and offers an approach well suited for this class of scientific applications.

Looking at literature studies on the exploitation of the hardware support, like HPCs, for gathering profiling data for detecting how different applications co-exist on the same hardware—and how well they use the memory hierarchy—we can find the proposals in [108]. In particular, these are oriented to determine whether different applications have (or not) a good sharing of the various components within the caching system. The outcoming data are used as a kind of profiling outcome, which can help establish the final deployment of the applications on the computing platform. However, they are not used as runtime support for automating the improvement of the job on the hardware components carried out by the applications.

Looking at works that rely on runtime optimisation capabilities, we also find solutions that attempt to optimise the CPU usage when multiple different tasks (e.g. computing and communication) need to be carried out [64, 58, 49]. These solutions still avoid the usage of HPCs, hence working at a higher level of analysis of the effects of the applications on the hardware.

The reliance on lower-level profiling data has been taken into account in recent works that attempt to optimise the energy usage when running

parallel/concurrent applications on multi-core machines [33]. These solutions are not oriented to consolidating multiple applications on the same hardware. Instead, they tackle the scenario where a single application with multiple threads with data dependencies (like transactional applications) is executed. In any case, the objective of these proposals only stands in controlling the ratio between performance and energy usage, with no additional optimisation in terms of, e.g., the percentage of time along which the CPU-core is not retiring executed instructions.

The objective of energy-usage reduction, in combination with the maintenance of close to optimal performance has also been studied by the side of the synchronisation support at the software level, like for the case of locking primitives [102]. In this approach, we still have a limit on the runtime support in terms of the type and amount of data being collected through hardware-level profiling facilities. In fact, the only low-level profiling data are related to the energy used by the hardware. In contrast, any other parameters used to optimise the synchronisation construct are based on performance indices evaluated at the software level. Furthermore, the low-level data related to energy usage are not exploited as an input to the optimisation process. Instead, they simply represent the audit concerning the actions executed at the level of the synchronisation support.

Compared to all these works, our approach is proactive in exploiting profiling data coming from multiple HPCs. Also, this exploitation is automated and targets creating execution timelines where gangs of applications are left to be CPU-dispatched by the operating system while others are temporarily blocked. As a matter of fact, our solution can be seen as orthogonal to several of these solutions, thus being open to co-existence.

5.2 The Consolidator infrastructure

We built our tool as a Linux Loadable Kernel Module (LKM) to simplify interaction with OS components and hardware control. Although its core is presented as a unique LKM object, it comprises three different parts: the *HPC driver*, the *profiler* and the *consolidator*. Rather than applying a module-stacking but a library-like technique as the baseline strategy. Consequently, both the HPC driver and the profiler expose functions marked as __*week*, which the consolidator redefines at compile time to connect all the elements and insert the analysis procedure. The consolidator represents the top level element and directly exploits the profiler capabilities to collect runtime information about desired processes. In compliance with custom logic, which we shall describe in Section 5.2.3, it ensures that only the selected group of processes runs in a specific time window. In such a way, it guides the OS scheduler and scans the workload evolution without ex-

ternal interference, also dealing with the collection of metrics that can't be attributed to a precise process due to design constraint (see Section 5.2.3). Figure 5.1 depicts the general structure of our tool and the relation among all the involved pieces.

5.2.1 The HPC driver

The HPC driver is the lowest element in the logical organisation of our architecture, which manages the interaction with the HPCs available in the system. It queries the machine capabilities upon module load by invoking a sequence of `cpuid` instructions, in order to calibrate all the required parameters to capture performance events.

To build a *consolidation profile*, i.e. to determine what are the best co-scheduling groups that allow maximising the performance of the applications avoiding incurring side effects from contention on the underlying hardware, we rely on runtime information collected from HPCs.

While it is possible to read information associated with the events directly from the HPCs, performing an analysis to support consolidation based on these raw data may be daunting. We have therefore implemented within our HPC driver logic to automatically apply the Top-Down Microarchitectural Analysis (TMA) [162] methodology. This methodology builds representative metrics for the underlying architecture targeting the analysis of software execution bottlenecks. Analysing the micro-architectural design makes it possible to define macro-areas where the bottleneck may lay. It divides the exploration into various levels, and the deeper level is taken into consideration, the more accurate is the inspection. At the moment, the TMA methodology spans four different levels. For example, an analysis carried out at the first level allows one to identify coarse-grain causes for performance bottlenecks, such as deeming an application as CPU front-end or back-end bound. Going deeper provides a more precise identification of the root cause, e.g. at level four, we can identify the bottleneck in the particular use of vectorised FPU instructions or in memory bandwidth limitations.

The TMA methodology requires observing many architectural events through HPCs, especially at the lowest levels. Unluckily, typical off-the-shelf architectures offer only a handful. Therefore, to implement the TMA methodology, it is necessary to multiplex the available HPCs massively, i.e. it is necessary to switch the architectural event being monitored repeatedly. Nevertheless, this multiplexing activity can severely impact both monitoring precision and cost, the former due to the possibility that a non-minimal number of relevant events is lost, the latter due to the continuous reconfiguration of the HPCs.

Our approach tries to limit the number of events to be monitored in two

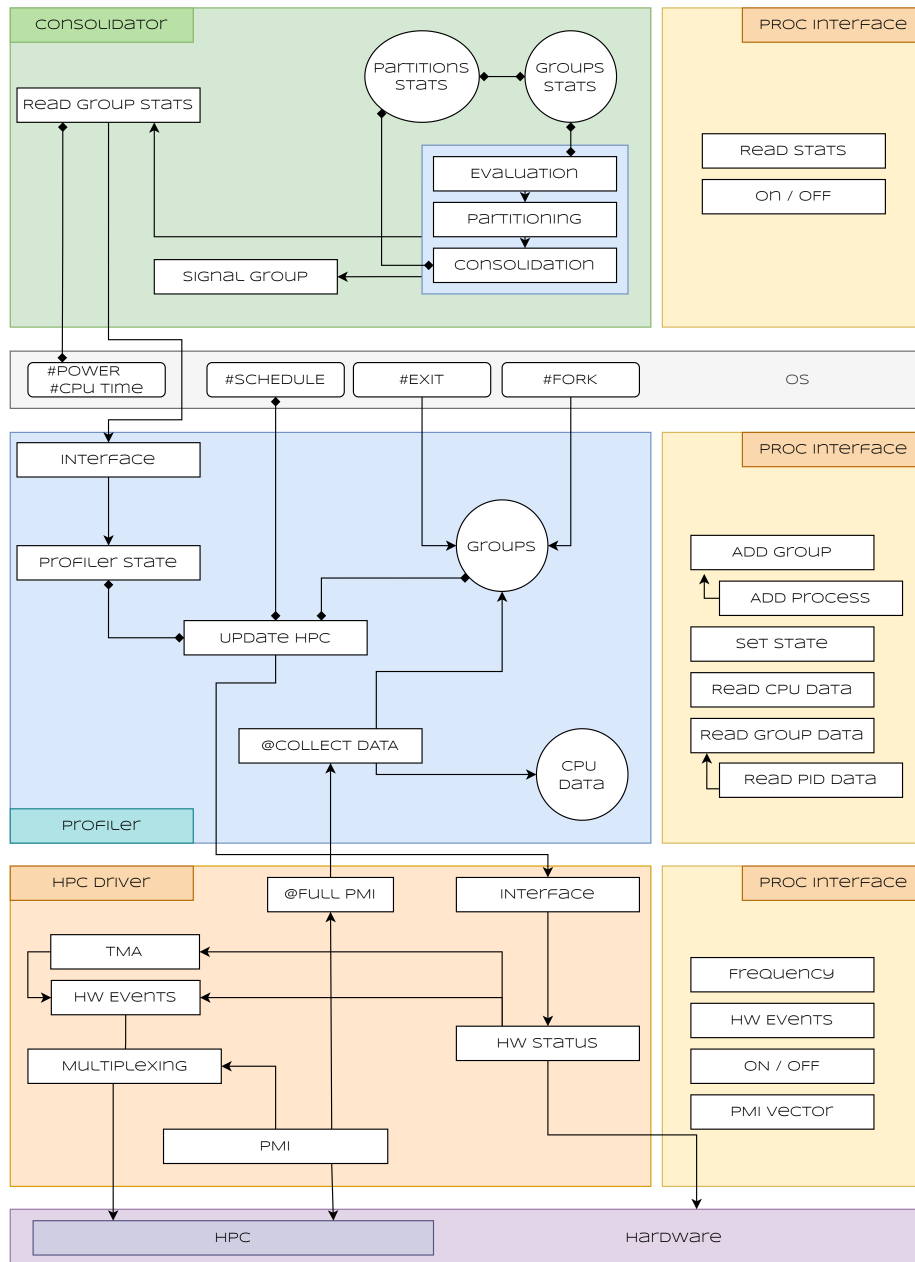


Figure 5.1: *High-level representation of the tool structure and interaction among the internal components.*

ways. First, the HPC driver can be configured to cut down the number of TMA levels after a defined value. Second, depending on the configuration provided by the Profiler, only the relevant branches of the TMA methodology are explored. In this way, the overall monitoring architecture is not overwhelmed by a large number of (possibly uncorrelated) events, and the number of reconfigurations of the HPCs is also reduced. Overall, this strategy improves the profiling accuracy and decreases its overhead simultaneously.

Microarchitectural data are collected by configuring only one of the available hardware counters in sampling mode to keep track of Time Stamp Counter (TSC) clock cycles[1]. This strategy allows us to generate a PMI at a system-level stable frequency, which is fundamental to tracking the system's evolution over time. Indeed, CPU cores can adapt their frequency at runtime, thus requiring a stable external source such as the TSC.

Conversely, all other HPCs are set in counting mode. Upon PMI generation due to the sampling counter overflow, all the raw values from other HPCs are read and packed with extra information such as CPU id, process ID and TSC value in a data structure. Furthermore, TMA metrics are computed. The generated data are directly passed to the profiler for further processing.

As a final note, the HPC Driver exposes in the `/proc` filesystem a set of pseudo-files that allow us to tune the measurement frequency and to enable/disable the overall monitoring process.

5.2.2 The Profiler

The main goal of the Profiler is to collect data coming from the HPC driver and allow userspace applications to register for profiling by writing their PID to a dedicated `/proc` pseudo-file. Every time a process registers itself, the profiler inserts it into a new *profiling group*, namely a container of processes. As mentioned, the HPC driver delivers data collected from HPCs stamped with the PID of the thread that was running when the events were collected. It is the role of the Profiler to aggregate all these data in a group profile.

Whenever a registered process forks or creates a new thread, the newly-generated `task_struct` is automatically registered within the same parent's profiling group. To intercept a fork invocation, the Profiler exploits the Linux Kernel static tracepoints[2]. The same concept is applied for thread

[1] On Intel processor, the *fixed* hardware counters #2 counts the number of reference cycles at the TSC rate. Still, TSC is a different counter that can be accessed by the RDTSC instruction.

[2] Linux Kernel static tracepoints are a lightweight support that allows attaching external code to hook points defined at compile time. Compared to other strategies, such as *ftrace* or *kprobes*, tracepoints provide a lower performance impact at the expense of

termination: whenever a thread exits, that event is detected, notified to the Profiler, and the thread is removed from its profiling group—this strategy also allows us to reclaim memory used for support data structures. Since a system-wide unique name identifies every group, a process can ask to join an existing group instead of defining a new one upon registration. In this way, it is possible to build complex performance profiles flexibly, considering any number of (concurrent) applications.

Whenever a process that is not registered for profiling is scheduled[3], the profiler switches off the HPC support and conversely turns it on when required. This strategy still targets an overhead reduction because we avoid collecting data unrelated to any profiled thread that the Profiler will eventually discard. Also, it reduces the number of PMIs that the underlying HPCs fire.

5.2.3 Finding the best Co-Scheduling

The component at the logical top of our architecture is the Consolidator. It exploits the profiler capabilities to collect runtime information about desired processes. As mentioned, its ultimate goal is to build and exploit a consolidation profile. This approach is based on its capability, built on top of standard Linux schedulers, to ensure that only the designated groups of processes run in a specific time window. Building an accurate consolidation profile is possible thanks to the underlying layers, which allow observing the workload evolution without external interference.

Current Linux kernel versions are shipped with the *cgroup* capability, that has been designed to group processes. Nevertheless, we have explicitly avoided relying on this feature because it does not allow to change the group structure at runtime, with a reasonable overhead. Consequently, we emulate process grouping by exploiting signals. In particular, we stop and resume specific processes at runtime, depending on what group(s) of processes are allowed to run. Therefore, enabling the scheduling of a group entails broadcasting the `SIG_CONT` signal to all of its members. Conversely, disabling a group is accomplished by sending a `SIG_STOP` signal. Determining the members of a group is done by directly interacting with the Profiler, which keeps track of group information provided via the `/proc` filesystem, and intercepts thread creation/termination.

The overall strategy enacted by the Consolidator is shown in Figure 5.2. It consists of different execution phases controlled by a high-resolution timer that periodically fires, invoking the scheduling management routine of the Consolidator.

---

flexibility.

[3]We rely on tracepoints also to detect a context switch event.

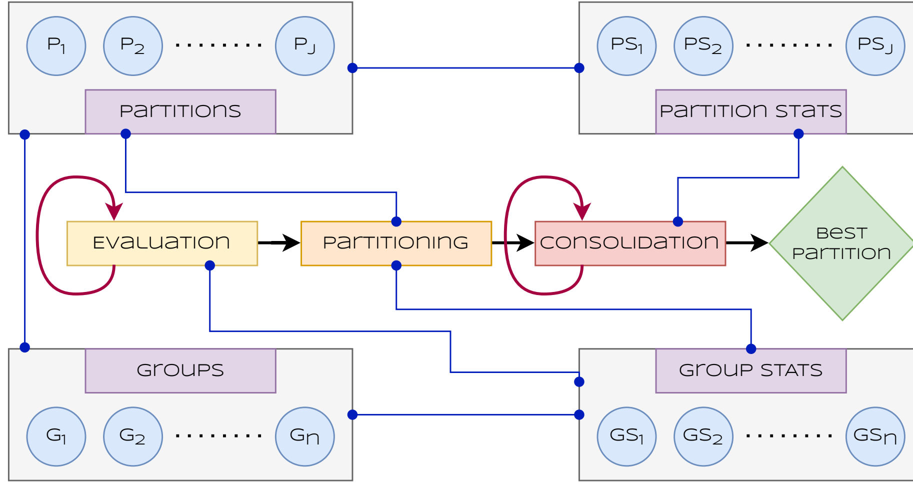


FIGURE 5.2: *Consolidator process to find the partition that provides the best score.*

The *evaluation* phase is the first step, in which each group is exclusively executed for a specified amount of time, defined by the system parameter τ. Next, the related CPU_{usage}, which is expressed as the percentage of the whole CPU time availability, is collected and stored within the group profile. That value constitutes the metric on which our approach builds all the possible schedule partitions in the subsequent phase.

The *consolidation* stage aims at finding the group co-schedule that provides the best score computed via Equation 5.1.

$$Score = \frac{CPU_{usage} * Useful_Work}{Energy} \tag{5.1}$$

Each of the involved variables represents the collected statistics during the consolidation phase. They are collected for each involved process (thanks to the HPC driver), aggregated into group profiles (thanks to the Profiler), and then in co-schedule statistics by the Consolidator.

CPU_{usage} is a global measure extracted by querying each core in the system, as shown in Listing 11. The kernel provides a per-CPU struct that holds the related core's time spent while processing or being in a specific state—the time domain is expressed in jiffies, and this value is updated at some "safe" points like context switches, interrupt management, and system calls invocation. Hence, we can accumulate both the used and total time for each processing unit deriving the CPU_{usage} as the relation of these two values.

Power consumption, defined as *Energy*, is another global measure. Contrarily to the previous one, it cannot be computed as the combination of single contributions. Instead, it is directly obtained by dedicated hardware

Listing 11 Routine to get cpu usage

```
1  void read_cpu_stats(u64 *used, u64 *total)
2  {
3      int i;
4      u64 _used = 0, _unused = 0;
5      for_each_online_cpu(i) {
6          struct kernel_cpustat kcpustat;
7          u64 *cpustat = kcpustat.cpustat;
8
9          kcpustat_cpu_fetch(&kcpustat, i);
10
11         _used   += cpustat[CPUTIME_USER];
12         _used   += cpustat[CPUTIME_NICE];
13         _used.  += cpustat[CPUTIME_SYSTEM];
14
15         _unused += cpustat[CPUTIME_IDLE];
16         _unused += cpustat[CPUTIME_IOWAIT];
17         _unused += cpustat[CPUTIME_IRQ];
18         _unused += cpustat[CPUTIME_SOFTIRQ];
19         _unused += cpustat[CPUTIME_STEAL];
20         _unused += cpustat[CPUTIME_GUEST];
21         _unused += cpustat[CPUTIME_GUEST_NICE];
22     }
23
24     *used = _used;
25     *total = _used + _unused;
26 }
```

probes which observe the whole chip activity. Consequently, we monitor the total system power consumption and collect data within the designated time window. Assuming that the scheduled groups are the sole active workloads (as in the evaluation phase), this technique estimates the associated power consumption with a high confidence level.

The last required parameter to compute the score is *Useful_ Work*, as seen from a micro-architectural perspective. In particular, this parameter captures the intrinsic dynamics of modern architectures compared to the actual instructions composing the application. From a general point of view, a modern processor can be split into *frontend* and *backend* units. The former is in charge of fetching instructions from memory and delivering them to the execution unit, while the latter performs the actual computation. Moreover, since we are dealing with out-of-order processors, two additional concepts must be introduced: *speculative activity* and *μops retirement*.

The *μOps retirement* parameter relates to the number of *μOps* effectively retired at each clock cycle[4], and expresses the percentage of time the machine is committing useful work. This value is computed per each process and then aggregated at upper levels. In other words, Equation 5.1 expresses a way to identify the partition that maximises both the machine throughput and utilisation while minimising energy consumption. These four compo-

[4]On most Intel processor models, the maximum number of retired $\mu Ops/clock_cycles$ is 4.

nents define the first TMA level, whose management, as anticipated, is a built-in capability of the HPC driver.

As a preliminary operation, we enumerate all the partitions of the group set which satisfy defined CPU utilisation constraints. Every subset that belongs to a partition should employ all the available computing power without exceeding the maximum machine availability. Ideally, such a value is 100% of the available computing resources. Still, to let our algorithm find a viable solution, we relax the problem constraints such that, given any subset S_i, its overall CPU utilisation should be:

$$\alpha \leq CPU_{usage}(S_i) \leq 2\alpha \tag{5.2}$$

Whenever $CPU_{usage}(S_i)$ is greater than 100%, the Consolidator scheduler simply relies on the underneath OS scheduler, i.e. it does not stop any thread. Given any partition, each subset represents a co-schedule, namely the set of groups placed into the run queue and executed concurrently. The problem can be seen as a variant of the partition problem, in which we find all the possible k-subset partitions for a given set, where k is the number of subsets inside the partition itself. To maximise the CPU utilisation while minimising the likelihood of exceeding the 100% value, we compute the starting value of k as:

$$K = \frac{\sum_{i=1}^{n} CPU_{usage}(G_i)}{\alpha} \tag{5.3}$$

All the k-partitions so generated are filtered out according to Equation 5.2. If the algorithm can produce any valid partition iteratively, it decreases k and restarts computing a solution. It is important to note that reducing each time the value of k leads, eventually, to the trivial 1-partition output, which represents the ordinary system's state in which all the groups are active and managed by the OS scheduler.

Data structures containing profiling information are then generated for each partition and co-schedule. Additionally, a partition having a unique subset with all the groups is created and pushed into the list to allow us to evaluate the plain OS scheduler activity.

In the *consolidation* phase, we individually execute and monitor each partition. We extract the first co-schedule S from the selected partition and place it into the run queue while all the other groups are still in a suspended state.

To guarantee fairness among all the co-schedules, they are activated for a time slot $t = \gamma\delta$, where γ is a configurable parameter, and δ is a weight factor that adjusts γ according to the required CPU utilisation. δ is computed as:

$$\delta = \frac{CPU_{usage}(S_j)}{\alpha} \tag{5.4}$$

Proving fairness for each partition P containing m subset is straightforward:

$$CPU_{usage}(P) = \sum_{j=1}^{m} CPU_{usage}(S_j) = \sum_{i=1}^{n} CPU_{usage}(G_i) \tag{5.5}$$

Thus, our solution extends the associated execution time when a group requires more than the available resources, ensuring consistency among the different partitions. Finally, after all the partitions have been profiled, the one delivering the best score is selected and employed as the active co-schedule strategy to manage the registered processes.

5.3 Experimental Results

To evaluate the effectiveness of our solution, we performed an experimental assessment on a machine equipped with an i7-10750H 6x (SMT) CPU and 16Gb of Ram, running Ubuntu 20.04 LTS with Linux Kernel 5.13.

In the first scenario, we characterise the machine score by executing some applications from the well-known stress-ng benchmark suite. According to the suite terminology, benchmarks are called stressors and provide individual workloads to concentrate the activity on a specific OS or hardware component. To consider a broad set of application characteristics, we selected *matrix-3d*, *bsearch*, *cpu*, *cache* and *fork*, described in Section 3.4, and the following stressors:

- *matrix*: perform different floating-point operations providing a valuable combination of memory and computing activity.
- *qsort*: perform sorting and search operation on integer array.
- *io*: continuously calls sync system call to commit buffer cache to disk.
- *hdd*: workers continually writing, reading and removing temporary files.
- *bigheap*: spawn different workers that grow their heaps by reallocating memory.

All the stressors were executed in parallel during this experimental evaluation, changing the number of active workers for each configuration. The SMT functionality was disabled during the whole testing phase to improve

the measurement stability, thus assigning to CPU_{usage} a maximum value related to 6-cores.

Figure 5.3a and Figure 5.3b show results for an 8-workers configuration. For the sake of space, we do not report all the computed partitions, showing only the four delivering the best scores, the four providing the worst scores, and the one associated with all the stressors running concurrently (OS in the plot).

Figure 5.3a depicts the high-level metrics plots we used to compute the final score for each of the active partitions. Conversely, Figure 5.3b shows the generated partition structure and the associated co-schedules. For each co-schedule, we highlight in bold the CPU_{usage} value, while the plain text indicates the stressor IDs within the co-schedule itself. Each ID can be mapped to the corresponding stressor by inspecting the OS partition. As we can see, the fairness property holds. Accordingly, each computed partition is executed for the same amount of time.

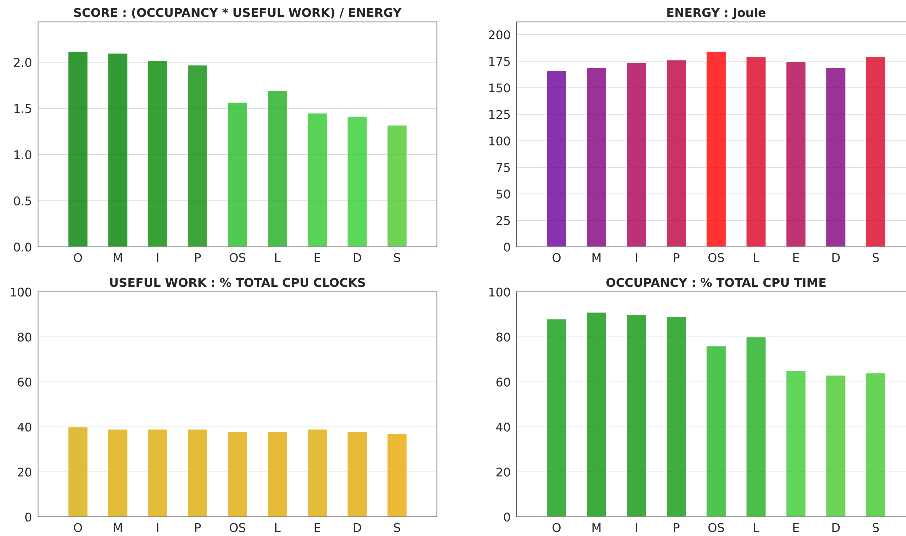
Figure 5.4, Figure 5.5, Figure 5.6 and Figure 5.7 depict results obtained by the different configurations evaluation, each of them labelled as $STRESS_x$, where x indicates the number id active processes for each individual stressor.

To evaluate our solution in a real-world context, we exploited the Parsec Benchmark Suite [20, 21], whose benchmarks show unique characteristics in terms of workloads, parallelisation granularity and data partitioning. In particular, we selected *swaptions*, *fluidanimate*, *dedup*, *freqmine*, *streamcluster* and *vips*.

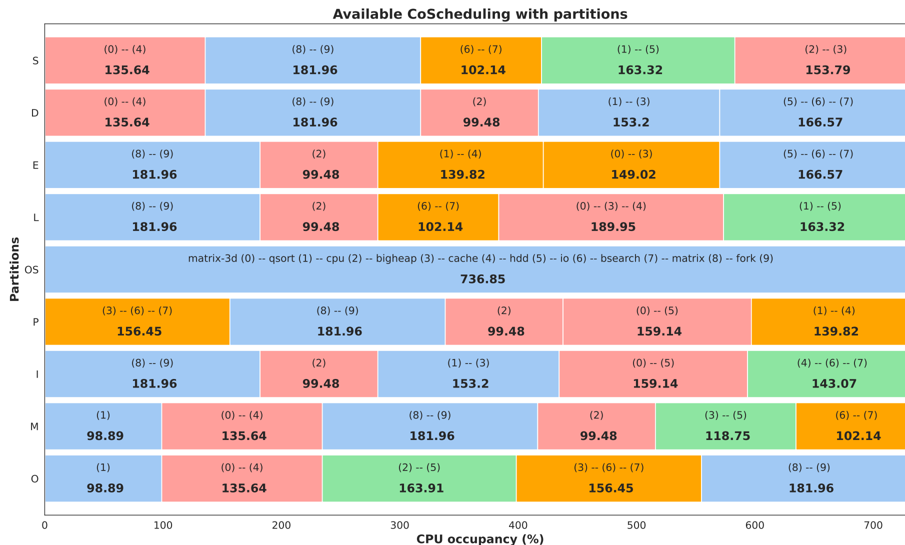
Contrarily to the previous experiment, we assessed a dedicated virtualised environment for each stressor to mimic the system activity in a cloud-like scenario.

Each workload was run into a virtual machine managed by the QEMU hypervisor. We adopted a lightweight and stripped version of Archlinux (kernel 5.10) as the operating system to minimize the measurement interference produced by the system activities and auxiliary components. Each VM was configured to execute a specific PARSEC benchmark at the startup phase, setting the number of the worker threads equal to the number of the virtualised processors. In such a way, selecting the desired configuration for an evaluation test was possible by merely modifying the QEMU parameters before launching the VM instance.

In such a scenario, our tool handles a QEMU instance as an individual entity that, at the registration phase, gets registered to the co-scheduling management process. Figure 5.8, Figure 5.9, Figure 5.10 and Figure 5.11 report the metrics values of Equation 5.1 for this experiment. In particular, for each configuration, we selected the best, and the worst partition computed by our consolidator and compared them with the plain operating



(A) *Consolidator process to find the partition that provides the best score. OS refers to the operating system scheduler activity without applying any partitioning, while the other labels indicate different computed partitions.*



(B) *Composition of the generated partitions and structure of each co-schedule. The OS partition contains the map among the stressors name and their IDs.*

FIGURE 5.3: *Consolidator process to find the partition that provides the best score. Stress-ng test with 8 parallel workers per stressor.*

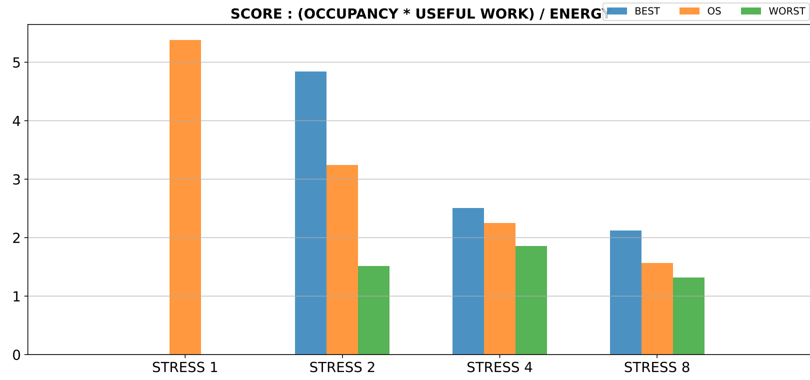


FIGURE 5.4: *Score of the different configurations of stress-ng workloads.*

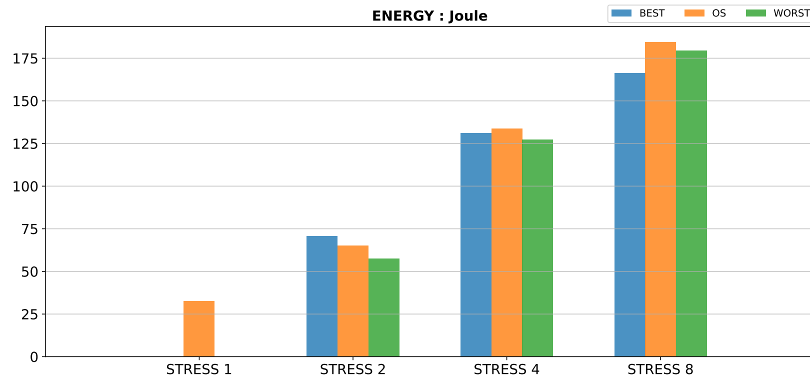


FIGURE 5.5: *Energy metric of the different configurations of the stress-ng workloads.*

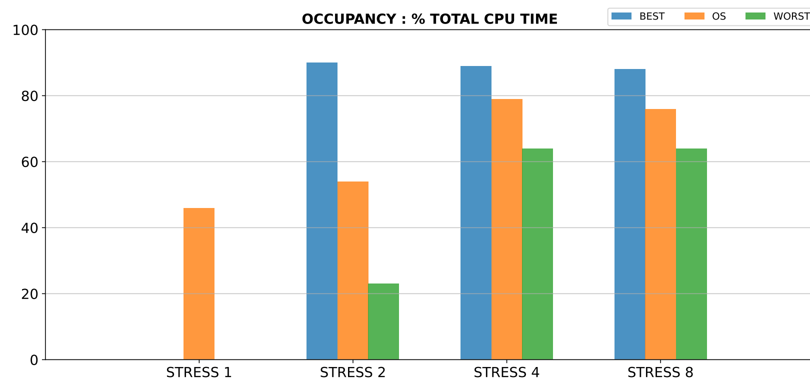


FIGURE 5.6: *CPU occupancy metric of the different configurations of the stress-ng workloads.*

system scheduler activity. Each run is labeled as S_x e P_y, where x refers to the number of parallel QEMU instances and y represents the number of

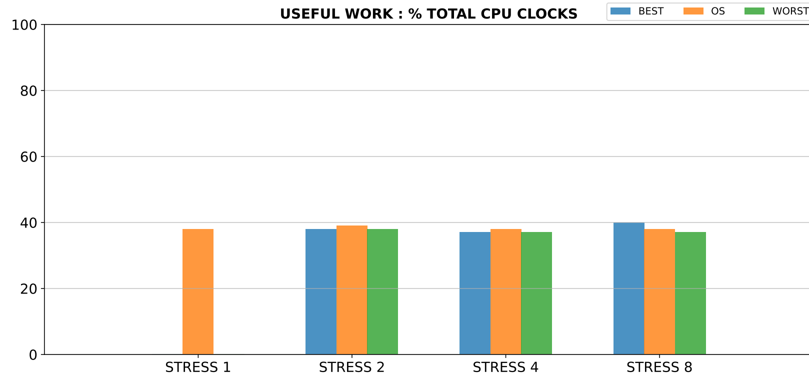


FIGURE 5.7: *Useful Work metric of the different configurations of the stress-ng workloads.*

virtual processors for that VM, namely the number of active processes.

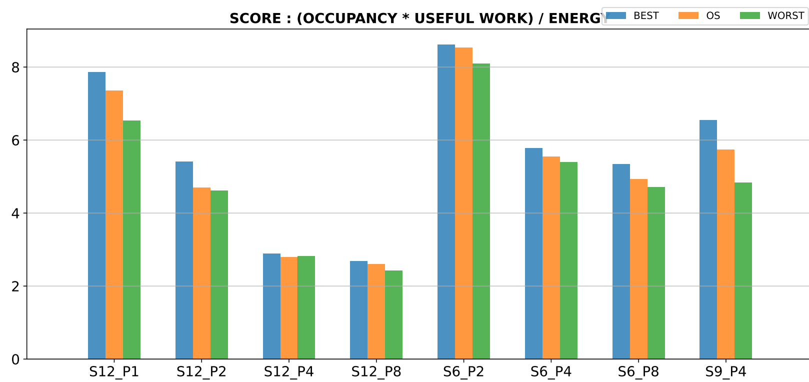


FIGURE 5.8: *Score of the different configurations of the QEMU-based experiment.*

As we can observe in Figure 5.8, we can always find a co-schedule with a better score compared to the one delivered by the traditional operating system scheduler. It is also important to note that such a score is not always directly influenced by the power consumption of the executed workload. Sometimes, the combination of the other two metrics has the most significant impact (see configuration S12_P2).

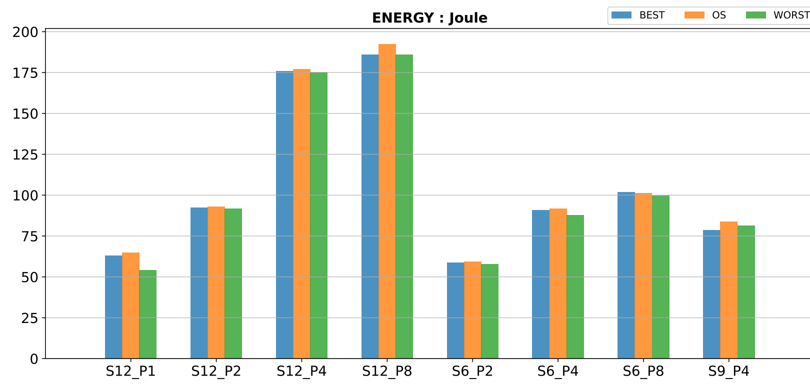


FIGURE 5.9: *Energy metric of the different configurations of the QEMU-based experiment.*

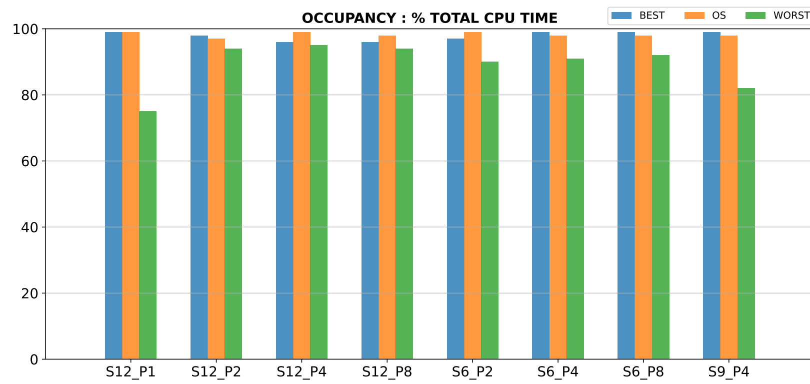


FIGURE 5.10: *CPU usage metric of the different configurations of the QEMU-based experiment.*

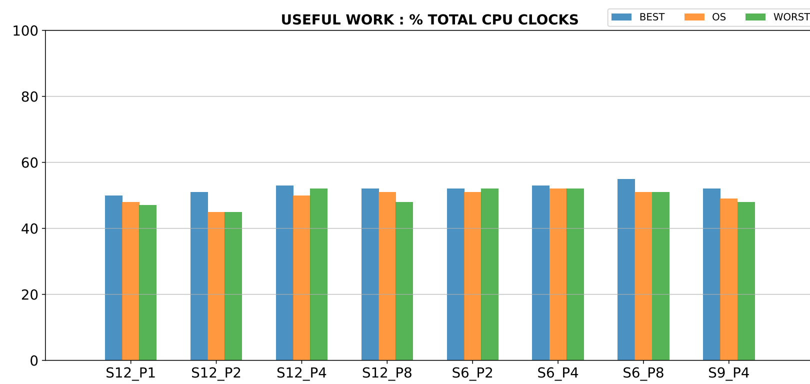


FIGURE 5.11: *Useful Work metric of the different configurations of the QEMU-based experiment.*

CHAPTER 6

Accelerating PDES via hardware-assisted incremental checkpointing

Speculative Parallel Discrete event Simulation (PDES) is known to be a core method for deliverying high performance of model execution [76], and for enabling the full exploitation of the available computing resources in both distributed and shared memory settings [14, 66]. At the same time, a core aspect to be considered when building speculative PDES platforms is the ability to reconstruct past simulation states whenever a causality violation—caused by wrong speculation paths—needs to be undone.

Several literature proposals exist, envisaging different state-reconstruction methodologies, which can be roughly classified as checkpoint-based [128] or reverse computing-based [29]—although in [31] a solution to mix the two strategies has been presented. In any case, the actual implementations of the state-reconstruction support mostly rely on pure software-based approaches, and do not exploit hardware-level operations that are nowadays commonly available in modern processors.

In this chapter we explore the alternative approach where the support for state reconstruction is devised as a hardware-assisted facility. Particularly, we focus on the checkpoint-based methodology, and present an architectural design of the speculative PDES engine where hardware-level program profiling capabilities offered by modern Intel processors are exploited to detect (with no software intervention) the state updates that occur while processing an event at the simulation object. This hardware-based detection is exploited to identify the portions of the object's state that have been updated along a sequence of events, which need to be logged to build a new incremental checkpoint. In this work, we explicitly target the x86-64 architecture and the Linux operating system, as a test case for the viability of our hardware-assisted checkpointing support.

One core aspect in our proposal is that the detection of memory updates occurs at the exact granularity of the machine-instruction that performs

the update. This provides a big advantage over memory-update detection actuated with other more traditional hardware-level mechanisms, such as the paging-firmware commonly exploited at the level of the operating system. In fact, the latter is known to work with page-based granularity—corresponding to 4KB in standard configurations on x86-64 processors—which can be clearly proven suboptimal for finer grain updates, possibly scattered across multiple operating-system pages.

We also note that our solution retains application transparency, since the activation of the hardware-level facility that enables memory update detection does not require any particular action by the user-defined event handler that implements the simulation logic used to process the events at the simulation objects. In fact, all the job of coordinating the hardware-level profiler and the execution of the event handler is carried out by the PDES runtime environment, which manages the activation of the application-level event handlers.

Clearly, detecting memory updates with no software intervention determines a drastic reduction of the cost of incremental checkpointing, thush making speculative PDES prone to delivering ever increasing speedup. Overall, our proposal is along the path of using hardware-level facilities as accelerators in the contexts of speculative PDES. However, this is done in an unconventional manner since we do not accelerate the execution of the event-handler logic—as it occurs when porting this logic to GPGPU architectures [96, 94]. Rather, we avoid the inclusion of additional machine instructions that would otherwise be needed to intercept the memory updates natively coded within the event handler by the application programmer. In other words, we virtualize the presence of these instructions replacing them via "accelerated" harware-level tasks.

The original contributions of this chapter can be summed up as follows:

- We propose a new way of adopting the Intel PEBS to provide support beyond application profiling.
- We enhance the system with an application-independent transparent memory tracing layer without applying software instrumentation.
- We provide experimental evidence of the effectiveness of our proposal when employed to support the execution of a synthetic model derived from the classical PHOLD.

6.1 Related Work

How to enable state reconstruction in speculative PDES in an efficient manner is a long-lived research topic. Literature proposals are very disparate

and tackle a wide spectrum of aspects. The proposals in [128, 134, 50, 142, 130] are agnostic of the way a single checkpoint is built; rather, they only need to know what is the (average) latency for taking the checkpoint in order to determine how frequently (or at what points along simulation time) checkpoints should be taken in order to optimize the tradeoff between the checkpointing cost and the state reconstruction cost—we recall that an un-checkpointed state needs to be recontructed by reloading the latest checkpoint preceeding that state and then reprocessing intermediate events. Essentially, these proposals provide performance models (sometimes used as run-time decision models) which do not directly address the issue of reducing the cost of each single checkpoint operation. Hence, they can be considered as orthogonal to what we propose in this chapter.

How to reduce the cost of the individual checkpoint operation has been tackled by other studies. [131] present a hardware-assisted solution where programmable DMA engines are used to implement data-copy operations which allow to fill checkpoint buffers with the current content of the simulation object state—hence offloading the checkpoint operation from the CPU. This solution does not offer the support for incremental checkpointing, since the DMA based data copy operation always stores the entire simulation object state, a limitation that appears to be relevant given the memory-wall phenomenon in modern processors. Also, it can be used only under the constraint that the simulation object state (which represents the source of the DMA operation) is stored in a contiguous memory buffer. Our porposal removes both these limtations since we enable incremental checkpointing and we support simulation objects' states that are scattered in memory.

Still by the side of hardware-assisted solutions, [137] present an architecture where incremental checkpointing in the context of HLA-based simulations is achieved transparently via operating-system services. However, the granularity according to which memory accesses are tracked to determine what portion of the state to log within the incremental checkpoint corresponds to an entire operating-system page. We avoid this limitation since our hardware-assisted mechansm for tracking memory updates operates at the memory-granularity of each single memory-write machine instruction.

[52] have proposed the "rollback-chip", which is a specialized hardware component used to store state variables according to a multiversion scheme. This enables keeping within this hardware component checkpointed versions of the state of the simulation object. State restoration is achieved by instructing the rollback chip to realign the live state to the selected checkpoint. Differently from this proposal we do not rely on specialized hardware. Instead, we exploit conventional faclities offered by Intel CPUs, which makes our proposal applicable in a wider spectrum of contexts.

As for software based implementations of incremental checkpointing, op-

timized approaches rely on (semi-)transparent instrumentation of the event-handler code to inject additional instructions used to identify state updates [158, 135, 124]. These approaches have been shown to work well especially with read intensive workloads, where the fraction of instructions that perform memory updates is relatively small. Our objective is the one of avoiding at all the instrumenting instructions and migrating the task of tracking state updates directly to the hardware. This will make incremental checkpointing viable even in scenarios with more write-intensive workloads, for which the cost of tracking memory updates via additional software instructions could not pay off—with respect to taking non-incremental checkpoints of the whole state of the simulation object.

Our work is (less closely) related to studies targeted at the exploitation of specific hardware capabilities (or accelerators)—such as GPGPUs or FPGAs—for improving the execution speed of PDES platforms [94, 160, 132]. Compared to these proposal, our objective is the one of accelerating the execution of specific tasks carried out by the PDES platform, while still keeping the application executing on a conventional CPU architecture.

Finally, our proposal is fully orthogonal to the solutions based on reverse computing, such as the proposal by [138] or the one by [31]. These proposals still rely on the usage of infrequent checkpoints to optimize the delivered performance. In fact, our hardware-assisted checkpointing architecture can be used as a black-box checkpointing support in such combined techniques.

6.2 The Hardware-assisted Incremental Checkpointing Architecture

The hardware-assisted incremental checkpointing architecture which we have devised is composed of two different parts. On the one hand, we must be able (thanks to the discussed hardware support) to detect what are the portions of the simulation state which are touched in write mode during the execution of simulation events targeted at each specific simulation object involved in the simulation run. On the other hand, we must be able to exploit and organize this information in a way that allows to effectively handle state saving and restore operations, carried out to recover from causal violations due to the speculative nature of the simulation. In this section, we discuss separately both aspects.

6.2.1 Hardware-assisted Memory Write Tracing

In our hardware-assisted incremental checkpointing architecture we have relied on the PEBS support. In particular, we have implemented a Linux

kernel module which exposes some services to the userspace PDES runtime environment to activate the tracing of memory write architectural events. The facilities offered by this kernel module are activated via proper `ioctl()` calls, towards a special device file which is created upon module load.

A first aspect to take into account when dealing with the tracing of memory write operations is that, despite the fact that the actual tracing is implemented via firmware facilities, a small overhead is anyhow introduced. This is related to the fact that the firmware has to pack PEBS records and write their content in main memory into the PEBS buffer, upon any memory access in write mode. This buffer is located in RAM, therefore writing PEBS records consumes a certain amount of memory bandwidth. On more recent NUMA architectures, the penalty associated with writing into memory could be also exacerbated by the fact that a CPU core might be required to write to memory banks associated with remote NUMA nodes. This scenario could also lead to interference with the activities of other cores, as the remote memory access is carried out by relying on inter-cache controller messages, as in the case of CPU interconnect based on the Intel QuickPath.

It is therefore fundamental to limit the tracing operation only to the execution of event handlers associated with the simulation model. In this way, any memory write operation carried out by the PDES runtime environment to support its internal scheduling, checkpointing, or any other housekeeping operation will be executed with no active tracing operation, thus significantly reducing the overall overhead. Nevertheless, a general-purpose PDES runtime environment must account for simulation events which can have any execution time granularity. It is therefore fundamental to devise a solution which can quickly activate/deactivate the tracing facility based on PEBS, therefore we have decided not to rely on an `ioctl()` call for this very specific activity. Indeed, although the transition to kernel mode is quite fast on modern architectures (thanks to the introduction of the `syscall` assembly instruction), the activation of the proper `ioctl()` handler involves the execution of several kernel-level software trampolines (one associated with the system call dispatcher, and one associated with the `ioctl()` dispatcher, which is part of the Virtual-File-System layer). Also, more modern versions of the Linux kernel, implement protection mechanisms against the Meltdown security attack—in particular the Kernel-level Page Table Isolation mechanism, KPTI—which involve a modification of the page table upon any mode switch from userspace to kernel space, thus introducing an additional overhead associated with the flush of the TLB. Overall, paying all these costs just to enable hardware-assisted incremental checkpoinfing while executing a single simulation event that could last only a handful of microseconds could be too much performance unwise.

To limit the overhead to activate/deactivate PEBS-based tracing, we rely on a dedicated software trap. Upon module load, we modify the Interrupt Descriptor Table (IDT) of every CPU core by nesting an ad-hoc interrupt handler on interrupt vector `0xf0`. This is a vector which is not used by the Linux kernel for any of its internal interrupt service routines. The routine which we hook at this vector is extremely simple: it simply writes into a MSR register a value to activate/deactivate the PEBS-based tracing. The PDES runtime environment can therefore execute an ad-hoc software trap (using the `int` assembly instruction) which generates an execution path of a few assembly instructions in kernel mode. This execution flow is also KPTI-compliant, and therefore introduces a very negligible overhead.

When the PEBS-based tracing support is active, the firmware will write PEBS records into the PEBS buffer. This buffer is allocated in kernel space upon module load. We use a buffer of 16 contiguous (in physical memory) 4KB pages, taken directly from the Linux buddy system. If the PEBS index reaches the PEBS-buffer occupancy threshold, the firmware will activate a second ad-hoc handler (also hooked in the IDT upon module load) which simply moves the current buffer into a pool of buffers to be consumed by the PDES runtime environment in userspace, and allocates a new buffer to be used as the PEBS buffer.

Another aspect to take into account is the time-sharing nature of the Linux operating system. In particular, the PEBS-based hardware mechanism is completely unrelated to the scheduling activities of the kernel. It is therefore perfectly possible that, while the PDES runtime environment is executing a simulation event, the time quantum allocated to it expires. In this scenario, the scheduler might give the CPU to a different thread, completely unrelated to the PDES simulation engine, with the PEBS-based support still active. The firmware writes in the PEBS buffer virtual addresses, and it is therefore possible that many samples will be associated with addresses which are either unmapped in the PDES simulation process, or are associated with buffers which are valid, yet not related with the simulation model. To avoid this phenomenon, the kernel module attaches an ad-hoc routine at the end of the execution flow of the scheduler of the Linux kernel—this is done by relying on the `kprobes` facilities offered by Linux. In particular, every time that the scheduler determines that a new thread should be scheduled on some CPU core, we check the pid of that thread against all the pids of the PDES runtime environment—these pids are registered at simulation startup via an ad-hoc `ioctl()` call. If the about-to-be-scheduled thread does not belong to the PDES runtime environment, we perform a write operation on a MSR to disable PEBS-based tracing. In the opposite case, we explicitly enable PEBS-based tracing. By relying on

this approach, we limit a possible system-wide performance penalty, and we avoid the introduction of any noise in the PEBS buffer, with respect to memory write operations not associated with the PDES run.

The last aspect which we have dealt with is how to transfer the information from the PEBS buffer to the userspace application. As mentioned, the PEBS buffer is composed of physically-contiguous pages of memory, which can be obtained only at kernel level—contiguousness is a requirement for the firmware to be able to correctly generate PEBS records. Moreover, the content of PEBS records is not completely of interest for the PDES runtime environment, which is interested only in the information associated with simulation state memory write operations, rather than in a full CPU snapshot. We have therefore implemented an `ioctl()` call which allows to retrieve a set of tuples in the form $\langle base\ address, size\rangle$, where *base address* is the initial address of the memory area touched in write mode by the simulation model, and *size* is the amount of bytes touched by the operation. This `ioctl()` call is similar in spirit to the `readdir()` Linux system call: it returns from kernel space a stream of structures describing a set of the aforementioned tuples, which can be used to construct the metadata used to later build the incremental log. This call essentially consumes the pool of PEBS buffers which have been filled during the execution of one or more simulation events—once the data from a buffer in the pool is completely transferred to the userspace PDES runtime environment, the buffer is returned to the buddy system.

To summarize, the actions which the PDES runtime environment executes to lever the PEBS-based memory-write tracing are:

1. before scheduling any simulation event to any simulation object, the hardware-assisted tracing is activated (thanks to a fast dedicated software trap via the `int 0xf0` assembly instruction);

2. when the event is completely executed, the hardware-assisted tracing is disabled (again via the `int 0xf0` assembly instruction);

3. the information about what memory areas have been accessed in write mode by one or multiple events are queried from the module, via an `ioctl()` call.

This simple scheme allows to identify what is the portion of the simulation state which has been updated while processing events.

6.2.2 Managing Incremental Checkpoints

The information retrieved from the kernel module should be used by the PDES runtime environment to build and manage incremental checkpoints.

To this end, we borrow the baseline approach from [124]. In this proposal, each simulation object's state is managed thanks to a userspace memory manager which serves dynamic memory allocations via `malloc()` calls. In particular, the PDES runtime environment relies on a set of bitmaps to describe the current state of each simulation object, in terms of buffer allocations and memory updates.

The PDES runtime environment uses the $\langle \textit{base address}, \textit{size} \rangle$ tuples retrieved via `ioctl()` to flag all the corresponding memory chunks in the dynamic memory allocator metadata as dirtied. This information is used to pack a checkpoint which only has the chunks that have been updated since the last checkpoint (and the associated metadata).

This checkpoint is then linked to the checkpoint queue, in event timestamp order, as in the traditional Time Warp proposal by [76]. We note that this approach allows for great flexibility in the checkpointing scheme. Indeed, we are only required to update the metadata describing what portions of the simulation state has been updated after the execution of every event. The actual incremental checkpoint can be taken also after the execution of any number of simulation events. In this sense, our proposal is perfectly compatible with all previous literature results based on the usage of checkpointing intervals (rather than checkpointing at each event). Indeed, it is possible to rely on sparse state saving [91, 17], or on any form of adaptive state saving [118, 134, 50, 141, 129, 130].

To reconstruct a previous simulation state upon a rollback operation, the chain of logs is backward traversed, starting from the first checkpoint appearing in the simulation time axis before the restoration point—this is again a classical means to support state restoration. From each incremental log that is found in the chain, the PDES runtime environment puts back in the live image of the simulation state all the chunks that are available and have not yet been restored in possible previous iterations. In this way, only the "newest" chunks are restored, and multiple memory write operation for the same chunk are not executed (this would be the case if the state was reconstructed by traversing the chain in reverse order). Also, all the metadata describing which chunks are currently in use are restored.

The iterative restore procedure stops when all the in-use chunks that have been dirtied are restored. Although in principles this could entail an indefinite number of iterative backward steps along the log chain, in practice the restore operation can be immediately finalized once we find a full log while backward re-traversing the log chain. In fact, all the in-use chunks that have not yet been restored are immediately available inside the full log for copy-back operations. Full logs can be explicitly interleaved to incremental ones, according to the original proposal by [124].

6.3 Experimental Results

We have implemented the hardware-assisted incremental checkpointing architecture described in Section 6.2 within the open source ROOT-Sim speculative PDES runtime environment [122]. All the simulations have been run on an octo-core Intel i5-8250U CPU, equipped with 16 GB of RAM, running Ubuntu 18.04 with Kernel 4.9.1.

To assess the effectiveness of our approach, we rely on a baseline configuration which is based on the more traditional pure software-based incremental state saving technique. In this baseline configuration, we have relied on a custom `gcc` plugin which, during the compilation of the simulation model, inserts in a completely transparent way ad-hoc function calls before every memory write instruction. In this way, the PDES runtime environment can be notified of the pending write operation, and it can flag any relevant metadata to mark a portion of memory as dirtied, since the last checkpoint. Both the hardware-based implementation and the software-based implementation rely on the incremental state saving mechanism proposed by [124].

To study the performance of the hardware-assisted incremental checkpointing architecture, we have relied on a synthetic benchmark derived from the well known PHOLD benchmark presented by [51], explicitly embedding parameterizable memory operations' patterns. In this benchmark, each simulation object executes fictitious events which only involve the advancement of the local simulation clock to the event timestamp. Every time that an event is executed, a new fictitious event is scheduled, destined to any simulation object, with a timestamp increment following some exponential distribution. Implementations of this benchmark have been already used in the literature to study the performance of incremental state saving approaches [158, 131]. The execution of an event includes a busy loop (which emulates a specific CPU delay for event processing, and hence a specific event granularity) and/or read/write mode access to a fictitious, memory contiguous state buffer of a given size S. Large values for S would mimic applications with large memory requirements. On the other hand, the spanning of read/write operations across the state buffer determines the specific locality inside the object state, associated with the event execution.

The second benchmark is a real-world application, namely the Personal Communication System (PCS) benchmark, which models a GSM mobile network. Each simulation object models the state's evolution of an individual hexagonal cell, and the whole set of cells provides wireless coverage to a square region of variable size. Each cell handles a parameterizable number N of wireless channels, which are modeled in a high-fidelity fashion via explicit simulation of power regulation and interference/fading phenomena, according to the proposal by [78].

Upon the start of a call, a call-setup record is instantiated via dynamically-allocated data structures, which is linked to a list of already active records within that same cell. Each record is released when the corresponding call ends or is handed off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the current call setup to achieve the threshold-level signal-to-interference ratio (SIR) value. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions (and related variations).

This application is highly parameterizable. Beyond the already mentioned number N of wireless channels per cell, the set of configurable parameters entails: τ_A, which expresses the inter-arrival time of subsequent calls to any target cell; $\tau_{duration}$, which expresses the expected call duration; τ_{change}, which expresses the residual residence time of a mobile device into the current cell. These parameters affect the *utilization factor UF* of available channels, expressed as $UF = \frac{\tau_{duration}}{\tau_A * N}$. This impacts the granularity of the events, since the more the busy channels, the more power-management records are allocated and consequently scanned/updated during the processing of different events. On the other hand, higher values of the channel utilization factor lead to higher memory requirements for the state image of individual LPs.

We have configured the model to use 16 simulation objects, and we have run the model on 4 concurrent worker threads. Each simulation object has been associated with a state of 16 KB, and we have varied in the range $[3KB, 15KB]$ the average amount of memory touched in write mode during the execution of each simulation event at each simulation object. Due to the scattered nature of the buffers, a higher amount of total memory accessed in write mode increases the number of memory writes to be intercepted (both in the software and the hardware-based case). However, we have adopted an approach where the update operations are based on memory-block writing machine instructions, such as `stos`, which gives rise to a configuration not unfavorable to the software based solution (since a single interception via software allows capturing both smaller and larger memory updates).

As a first consideration, the duration of checkpoint/restore operations is not directly affected by using hardware- or software-based tracing of the memory-write operations. In particular, when updating 15 KB of simulation state per event, in both configurations we have observed an average checkpoint creation time on the order of 0.5 μs, and a recovery time on the order of 4.5 μs. Therefore, the data that we compare in the remainder of this section exactly allows to capture the effects on memory tracing of the

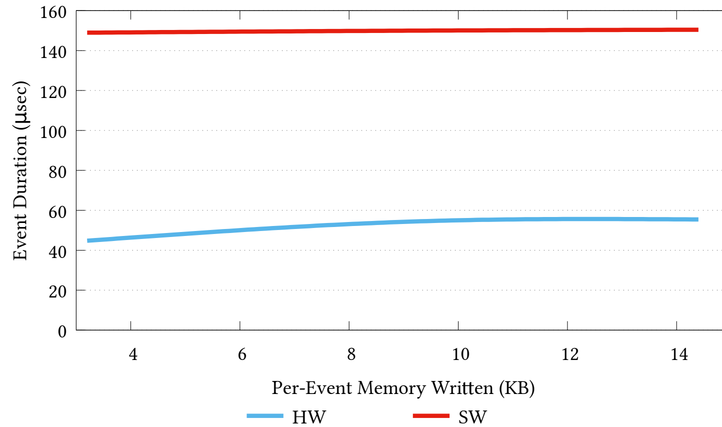


FIGURE 6.1: *Average Event Duration.*

two different supports (hardware vs software). In Figure 6.1 we report the average duration (in μs) of a simulation event when varying the size of the memory accesses. From the result, we can observe that the software-based solution (referred to as SW in the plots) is always outperformed by the hardware-assisted solution (HW in the plots)—although as discussed above it is not penalized by larger memory updates given the memory-block approach of writing machine instructions. This is an expected result, because (as we have discussed) the hardware-assisted solution relies on highly optimized firmware operations to trace a single memory access. On the other hand, for the same operation, the software-based solution requires activating a dedicated software function. This has also the drawback of requiring to save the content of several CPU registers for the execution of the model to continue correctly.

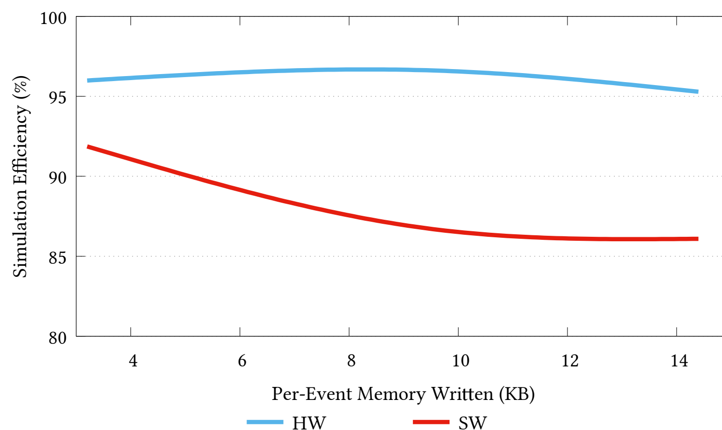


FIGURE 6.2: *Simulation Efficiency.*

This result is also related to the data reported in Figure 6.2. In particular, the reliance on a highly-optimized firmware increases the overall efficiency of the simulation—a reduced number of rollbacks is always observed when relying on the hardware-assisted solution. We believe that this phenomenon is related to a secondary effect on the usage of caches, ultimately related to the variance of the event execution times. Indeed, PEBS buffers are always contiguous in memory, and therefore accessing them to store PEBS samples will likely have a more circumscribed negative effect on the data used by the models and on the variance in the event execution time. Therefore, the reduction of variance of the duration of an event is also strengthened, an effect which is already known in the literature to have a possible positive effect on the rollback probability.

CHAPTER 7
Conclusions

Predicting the system response when dealing with highly parallel heterogeneous workloads represents a more and more pervasive challenge, quickly expanding from high-performance infrastructures to commodity hardware. Consequently, performing static-defined decisions might not be able to catch dynamic interactions stemming from non-deterministic complex environments. Sometimes, such integrated routines not only may fail to accomplish the desired result, but even introduce a performance slowdown by applying hypotheses that do not match the actual scenario. Furthermore, any attempt at software optimization would hold just for the current architecture with a bias toward the provided system configuration, which, in a widespread scenario, entails the generation of the execution trace in an idle system to lower any possible external interference. Indeed, even though the best-compiled versions of several programs are running, their coexistence may lead to some resources contention (since these are limited and shared) generating unpredictable effects and possible performance drawbacks.

Shifting to a dynamic strategy requires dealing with stricter constraints, making the overall process more challenging. On the one hand, system resources are generally shared among monitoring instruments and monitored processes requiring dynamic adjustments to avoid the entire system collapse caused by mistaken configuration or application behaviour changes. On the other hand, perturbing the target workload execution may generate polluted information leading decision-makers to wrong actions, as it is not well representing the runtime state.

To the best of our knowledge, Performance Monitor Units represent the most suitable technology to enhance profilers with low-overhead and transparent instrumenting capabilities, also delivering detailed hints from a hardware-level point of view. Still, their adoption is quite limited and existing tools are not efficient enough to be activated in production contexts.

One of the goals of this work has been to investigate the root of such inefficiency and devise alternative techniques to overcome such a problem. Despite its lightweight nature, this support is often coupled with software-level metrics or more general-purpose data collection techniques, mainly fo-

cusing on richer functional capabilities to fulfil offline analysis than prompt data delivery for online consumption. Unlike traditional approaches, we have discussed the performance impact of the various monitoring strategies, and our experimental assessment showed that many of these proposals can have a negligible impact on the application's performance profile under monitoring.

We also claim that the monitoring step shouldn't be agnostic to data processing, especially when conducted concurrently in an online fashion. Cohesion between the two phases is the crux to reaching better efficiency as it can regulate data generation without producing useless information and wasting disk space, memory bandwidth and computational resources.

At the same time, an important aspect to consider is the marked difference between the diagnosis we would like to perform and the related problem domain, which does not always converge to the same point. In particular, understanding problem characteristics and projecting them onto the underlying machine design effectively pinpoint the proper resource provisioning and optimal settings for the analysis. Such a process demands a deep knowledge of the used micro-architecture, which may discourage the exploitation of such supports or even produce misleading data. We firmly believe that combining raw events to define a high-level representation of hardware activities, like the one provided by the Top-Down Micro-architectural Analysis methodology, is the key to enforcing the analysis portability and its adoption curve.

We have presented our system-wide profiling infrastructure, an independent kernel module that smoothly accesses and manages PMUs facilities and automatically performs filtering and data elaboration to accomplish required operations. Furthermore, it directly links its features to OS core elements to broaden its sphere of action, and the data processing strategy can be easily adapted without dealing with lower-level details, thanks to its *modularity*. On the other hand, it ensures the *adaptability* principle by allowing ranging the number of applications subjected to monitoring activities, thus providing the ability to target a single application, a set of programs, or even the whole system, including kernel components. Moreover, our solution fosters adaptability by defining application-level hooks and exploiting augmented kernel-level facilities, providing mechanisms to trigger operations leading to accurate monitoring and timely decisions.

The experience we gained during this journey tells us that the methodology we followed, coupled with the techniques we described, is promising and can be applied unconditionally to various operational contexts. In particular, in this work, we have shown how our research can effectively assist in runtime contexts related to security reinforcement and system consolidation.

We have presented a system-wide detection system of cache side-channel attacks based on measures taken from HPCs, which, in turn, combines to build higher-levels decision metrics to deem a (multi-threaded) process as malicious. Moreover, we have coupled our detection system with mitigation actions, both for side-channel attacks and transient execution-based attacks relying on cache side channels to leak data. These mitigations are activated on a fine-grain basis at runtime, as opposed to scenarios where security-oriented tasks are carried out by default independently of the trustworthiness level of the running applications. The metrics we have devised allowed us to detect attacks with a negligible percentage of false positives and no false negatives on different x86 Intel CPUs and a comprehensive set of benchmark applications.

On the other hand, by leveraging the modular facilities of our infrastructure, we have assessed a new technique to drive the Operating System in the quest for the processes' placement. We have combined software and hardware metrics to derive a new methodology and classify different co-schedules according to the system's activity and the hardware feedback. Furthermore, the online exploration phase doesn't affect the slice of time the OS scheduler would assign to each involved process, assuring fairness and flexibility even within volatile environments. The experimental phase has shown that our strategy can always find a higher-score processes grouping than the solution delivered by the traditional operating system scheduler.

Additionally, we have explored the effectiveness of hardware profiling facilities in speeding up functional software procedures. In particular, we have exploited the Intel PEBS feature as a hardware accelerator for memory tracing support and evaluated its effectiveness in PDES runtime environments to support incremental state saving by intercepting memory-write operations performed by simulation models. Finally, we demonstrate the viability of PMUs in reducing the runtime and the intrusiveness of the incremental state saving support, traditionally associated with software instrumentation.

As future work, we plan to extend the application of self-tuning techniques to the profiling components themselves. As discussed in this thesis, the sampling frequency must be regulated according to the current execution context, and indeed this highly depends on both analysis type and goal. Moreover, performing a specific diagnosis requires a certain resource quantity, which is strictly related to system capabilities and problem specifications. It clearly states the need to identify the achievable profiling activity density without affecting the goodness of the generated data, and such a value cannot be effortlessly statically computed due to the high relation with the aforementioned variable parameters. We also wish to invest more effort in the facilities' assessment of profiling supports, concerning both their evolution and new releases. For instance, the concept of the heterogeneous-

cores design is not bounded to mobile chips anymore, but it states the new road-map for the upcoming products generation for different vendors, demanding the adaption of PMUs to fit new architectures, also delivering new supports such as the emerging Intel Thread Director [67].

Bibliography

[1] Aciiçmez, O., and Schindler, W. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In *Topics in Cryptology*, vol. 4964 of *LNCS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 256–273.

[2] Adhianto, L., Banerjee, S., Fagan, M. W., Krentel, M., Marin, G., Mellor-Crummey, J. M., and Tallent, N. R. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience 22*, 6 (2010), 685–701.

[3] Advanced Micro Devices. *AMD64 Technology Lightweight Profiling Specification*, August 2010.

[4] Advanced Micro Devices. *Revision Guide for AMD Family 10h Processors*, March 2012.

[5] Advanced Micro Devices. *AMD64 Technology AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions Publication No. Revision Date*, December 2017.

[6] Advanced Micro Devices. *Revision Guide for AMD Family 17h Models 30h-3Fh Processors*, September 2019.

[7] Akiyama, S., and Hirofuchi, T. Quantitative evaluation of intel PEBS overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS@HPDC 2017, Washingon, DC, DC, USA, June 27 - 27, 2017* (2017), pp. 3:1–3:8.

[8] AMD. Indirect Branch Control Extension. Tech. rep., AMD, 2018.

[9] AMD. Processor Programming Reference (PPR) for AMD Family 17h 01h,08h, Revision B2 Processors, 2019.

[10] Ammons, G., Ball, T., and Larus, J. R. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices 32*, 5 (May 1997), 85–96.

[11] ARM Holdings. *Processor Cortex®-A53 MP Core Software Developers Errata Notice*, 2019.

[12] ARM Holdings. *Processor Cortex®-A77 MP 074 Software Developers Errata Notice*, 2020.

[13] Bare, K. A., Kavulya, S., and Narasimhan, P. Hardware performance counter-based problem diagnosis for e-commerce systems. In *2010 IEEE Network Operations and Management Symposium - NOMS 2010* (April 2010), pp. 551–558.

[14] Barnes, P. D., Carothers, C. D., Jefferson, D. R., and LaPre, J. M. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS '13* (2013), p. 327.

[15] Barr, T. W., Cox, A. L., and Rixner, S. Translation caching: skip, don't walk (the page table). In *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10* (New York, New York, USA, 2010), ACM Press, pp. 48–59.

[16] Basu, K., Hussain, S. S., Gupta, U., and Karri, R. COPPTCHA: COPPA tracking by checking Hardware-Level activity. *Transactions on Information Forensics and Security 15* (2020), 3213–3226.

[17] Bellenot, S. State skipping performance with the Time Warp operating system. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation* (1992), PADS, pp. 53–64.

[18] Bhatia, S., Kumar, A., Fiuczynski, M. E., and Peterson, L. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 103–116.

[19] Bhattacharya, S., and Mukhopadhyay, D. Who Watches the Watchmen?: Utilizing Performance Monitors for Compromising Keys of RSA on Intel Platforms. In *Cryptographic Hardware and Embedded Systems*, T. Güneysu and H. Handschuh, Eds., vol. 9293 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2015, pp. 248–266.

[20] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008* (2008), A. Moshovos, D. Tarditi, and K. Olukotun, Eds., ACM, pp. 72–81.

[21] Bienia, C., and Li, K. Parsec 2.0: A new benchmark suite for chip-multiprocessors.

[22] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel*, 3 ed. O'Reilly & Associates Inc, Sebastopol, CA, Stati Uniti, Nov. 2005.

[23] Browne, S., Deane, C., Ho, G., and Mucci, P. Papi: A portable interface to hardware performance counters.

[24] Brumley, B. B., and Hakala, R. M. Cache-Timing Template Attacks. In *Advances in Cryptology*, M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2009, pp. 667–684.

[25] Brunella, M. S., Bianchi, G., Turco, S., Quaglia, F., and Blefari-Melazzi, N. Foreshadow-VMM: Feasibility and Network Perspective. In *Proceedings of the 2019 Conference on Network Softwarization* (jun 2019), NetSoft, IEEE, pp. 257–259.

[26] Bruska, J., Blasingame, Z., and Liu, C. Verification of OpenSSL Version via Hardware Performance Counters. In *Disruptive Technologies in Sensors and Sensor Systems* (may 2017), R. D. Hall, M. Blowers, and J. Williams, Eds., SPIE, p. 102060A.

[27] Calandrino, J. M., and Anderson, J. H. On the design and implementation of a Cache-Aware multicore Real-Time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems* (Piscataway, NJ, USA, July 2009), ECRTS, IEEE, pp. 194–204.

[28] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., Von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the 28th USENIX Security Symposium* (2019).

[29] Carothers, C. D., Perumalla, K. S., and Fujimoto, R. M. Efficient Optimistic Parallel Simulations Using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation 9*, 3 (jul 1999), 224–253.

[30] Chiappetta, M., Savas, E., and Yilmaz, C. Real Time Detection of Cache-based Side-channel Attacks using Hardware Performance Counters. *Applied Soft Computing 49* (dec 2016), 1162–1174.

[31] Cingolani, D., Pellegrini, A., and Quaglia, F. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. *ACM Transactions on Modeling and Computer Simulation 27*, 2 (jul 2017), 1–26.

[32] Compaq. *Alpha 21164 Microprocessor - Hardware Reference Manual.* 1998.

[33] Conoci, S., Di Sanzo, P., Pellegrini, A., Ciciani, B., and Quaglia, F. On power capping and performance optimization of multithreaded applications, 2021.

[34] Copos, B., and Murthy, P. InputFinder: Reverse Engineering Closed Binaries using Hardware Performance Counters. In *Proceedings of the 5th Workshop on Program Protection and Reverse Engineering Workshop* (New York, New York, USA, 2015), PPREW, ACM Press, pp. 1–12.

[35] Corbet, J. Taming STIBP. Tech. rep., LWN.net, 2018.

[36] Cruz, E. H. M., Diener, M., Pilla, L. L., and Navaux, P. O. A. Online thread and data mapping using a Sharing-Aware memory management unit. *ACM Trans. Model. Perform. Eval. Comput. Syst. 5*, 4 (Jan. 2021), 1–28.

[37] Das, S., James, K., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F. A flexible framework for expediting bug finding by leveraging past (Mis-)Behavior to discover new bugs. In *Annual Computer Security Applications Conference* (New York, NY, USA, Dec. 2020), ACSAC '20, Association for Computing Machinery, pp. 345–359.

[38] Das, S., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy* (may 2019), SP, IEEE, pp. 20–38.

[39] Dashti, M., Fedorova, A., Funston, J. R., Gaud, F., Lachaize, R., Lepers, B., Quéma, V., and Roth, M. Traffic management: a holistic approach to memory placement on NUMA systems. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013* (2013), pp. 381–394.

[40] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. On the Feasibility of Online Malware Detection with Performance Counters. In *Proceedings of the 2013 International Symposium on Computer Architecture* (New York, New York, USA, 2013), ISCA, ACM Press, pp. 559–570.

[41] Di Gennaro, I., Pellegrini, A., and Quaglia, F. OS-Based NUMA optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (May 2016), IEEE, pp. 291–300.

[42] Diener, M., Cruz, E. H. M., Alves, M. A. Z., Navaux, P. O. A., Busse, A., and Heiss, H.-U. Kernel-Based thread and data mapping for improved memory affinity. *IEEE Transactions on Parallel and Distributed Systems 27*, 9 (Sept. 2016), 2653–2666.

[43] Diener, M., Madruga, F., Rodrigues, E., Alves, M., Schneider, J., Navaux, P., and Heiss, H.-U. Evaluating thread placement based on memory access patterns for multi-core processors. In *Proceedings of the 12th International Conference on High Performance Computing and Communications* (Piscataway, NJ, USA, Sept. 2010), HPCC, IEEE, pp. 491–496.

[44] Disselkoen, C., Kohlbrenner, D., Porter, L., and Tullsen, D. Prime+Abort: A timer-free high-precision L3 cache attack using intel TSX. In *Proceedings of the 26th USENIX Security Symposium* (2017).

[45] Division, I. M. *PowerPC 604 RISC Microprocessor User's Manual*. 1994.

[46] Dobson, M., Gaughen, P., Hohnbaum, M., and Focht, E. Linux support for NUMA hardware. In *Proceedings of the 2003 Ottawa Linux Symposium* (Ottawa, ON, Canada, 2003), vol. 2003, Ottawa Linux Symposium.

[47] Drongowski, P. J., Team, A. C., and Center, B. D. An introduction to analysis and optimization with amd codeanalyst performance analyzer. *Advanced Micro Devices, Inc* (2008), 1–20.

[48] Eranian, S. Perfmon2: a flexible performance monitoring interface for linux. In *Proc. of the 2006 Ottawa Linux Symposium* (2006), pp. 269–288.

[49] Faltelli, M., Belocchi, G., Quaglia, F., Pontarelli, S., and Bianchi, G. Metronome: adaptive and precise intermittent packet retrieval in DPDK. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (New York, NY, USA, Nov. 2020), CoNEXT, ACM, pp. 406–420.

[50] Fleischmann, J., and Wilsey, P. A. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation* (1995), IEEE Computer Society, pp. 50–58.

[51] Fujimoto, R. M. Parallel Discrete Event Simulation. *Communications of the ACM 33*, 10 (oct 1990), 30–53.

[52] Fujimoto, R. M., Tsai, J. J., and Gopalakrishnan, G. Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp. *IEEE Transactions on Computers 41*, 1 (1992), 68–82.

[53] Garcia-Serrano, A. Anomaly Detection for Malware Identification using Hardware Performance Counters.

[54] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is dead: Long live KASLR. In *Engineering Secure Software and Systems* (2017), Springer International Publishing, pp. 161–176.

[55] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is Dead: Long Live KASLR. In *Lecture Notes in Computer Science*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds., vol. 10379 of *LNCS*. 2017, pp. 161–176.

[56] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *Lecture Notes in Computer Science*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., vol. 9721 of *LNCS*. nov 2016, pp. 279–299.

[57] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 Symposium on Security and Privacy* (may 2011), IEEE, pp. 490–505.

[58] Gureya, D., Neto, J., Karimi, R., Barreto, J., Bhatotia, P., Quema, V., Rodrigues, R., Romano, P., and Vlassov, V. Bandwidth-Aware page placement in NUMA. In *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium* (Piscataway, NJ, USA, May 2020), IPDPS, IEEE, pp. 546–556.

[59] Hansen, D. Hansen's KPTI Patch, 2017.

[60] Hassan, S., Al-Jumeily, D., and Hussain, A. J. Autonomic computing paradigm to support system's development. In *Proceedings of the 2nd International Conference on Developments in eSystems Engineering* (Piscataway, NJ, USA, Dec. 2009), DESE, IEEE, pp. 273–278.

[61] Helali, L., and Omri, M. N. A survey of data center consolidation in cloud computing systems. *Computer Science Review 39* (Feb. 2021), 100366.

[62] Herath, N., and Fogh, A. These are not your grand daddy's cpu performance counters. In *Black Hat USA (BH-US)* (2015).

[63] Herlihy, M., and Shavit, N. On the nature of progress. In *Principles of Distributed Systems*, A. Fernàndez Anta, G. Lipari, and M. Roy, Eds., vol. 7109 of *Lecture Notes in Computer Science*. Springer International Publishing, Berlin Heidelberg, Germany, 2011, pp. 313–328.

[64] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies* (New York, NY, USA, Dec. 2018), CoNEXT, ACM, pp. 54–66.

[65] Hund, R., Willems, C., and Holz, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Proceedings of the 2013 Symposium on Security and Privacy* (may 2013), IEEE, pp. 191–205.

[66] Ianni, M., Marotta, R., Cingolani, D., Pellegrini, A., and Quaglia, F. The Ultimate Share-Everything PDES System. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (New York, New York, USA, 2018), PADS, ACM Press, pp. 73–84.

[67] INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B and 3C): System Programming Guide.* 2022.

[68] INTEL, E. *Speedstep® technology for the intel® pentium® m processor.* 2004.

[69] INTEL CORPORATION. Intel Analysis of Speculative Execution Side Channels. Tech. rep., Intel, 2018.

[70] INTEL CORPORATION. *7th and 8th Generation Intel® Core™ Processor Family - Specification Update - Supporting 7th Generation Intel Core™ Processor Families based on Y/U/H/S-Processor Line, Y/U With iHDCP2.2-Processor Line , Intel Pentium Processors , Intel Celeron Processor and Intel Xeon E3-1200 v6 Processors.Supporting Intel Core™ X-Series Processor Families – 7740X, 7640X Supporting 8th Generation Intel Core™ Processor Family for U 4-Core Platforms, formerly known as Kaby Lake-R and Y 2-Cores, formerly known as Amber Lake Platform*, November 2019.

[71] INTEL CORPORATION. 8th and 9th Generation Intel Core Processor Family. Tech. rep., Intel, 2019.

[72] INTEL CORPORATION. *10th Generation Intel® Core™ Processor Families - Specification Update - Supporting 10th Generation Intel ® Core™ Processor Families, Intel ® Pentium TM Processors, Intel® Celeron® Processors for U/Y Platforms, formerly known as Ice Lake*, August 2020.

[73] IRAZOQUI, G. *Cross-core Microarchitectural Side Channel Attacks and Countermeasures.* PhD thesis, Worcester Polytechnic Institute, 2017.

[74] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – And its application to AES. In *Proceedings of the 2015 Symposium on Security and Privacy* (2015).

[75] J DRONGOWSKI, P. Instruction-based sampling: A new performance analysis technique for amd family 10h processors.

[76] JEFFERSON, D. R. Virtual Time. *ACM Transactions on Programming Languages and Systems 7*, 3 (jul 1985), 404–425.

[77] KADIR, M. F. A., WONG, J. K., AB WAHAB, F., BHARUN, A. F. A. A., MOHAMED, M. A., AND ZAKARIA, A. H. Retpoline Technique for Mitigating Spectre Attack. In *Proceedings of the 6th International Conference on Electrical and Electronics Engineering* (2019), ICEEE.

[78] KANDUKURI, S., AND BOYD, S. Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications. *IEEE Transactions on Wireless Communications 1*, 1 (2002), 46–55.

[79] Kayaalp, M., Abu-Ghazaleh, N., Ponomarev, D., and Jaleel, A. A High-resolution Side-channel Attack on Last-level Cache. In *Proceedings of the 53rd Annual Design Automation Conference* (New York, New York, USA, 2016), ACM Press, pp. 1–6.

[80] Kazdagli, M., Reddi, V. J., and Tiwari, M. Quantifying and Improving the Efficiency of Hardware-based Mobile Malware Detectors. In *Proceedings of the 49th Annual International Symposium on Microarchitecture* (oct 2016), MICRO, IEEE, pp. 1–13.

[81] Kephart, J. O., and Chess, D. M. The vision of autonomic computing. *Computer 36*, 1 (Jan. 2003), 41–50.

[82] Kim, T., Peinado, M., and Mainar-Ruiz, G. StealthMem: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium* (2012).

[83] King, C. I. Stress-ng: A stress-testing Swiss army knife, 2017.

[84] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy* (2019).

[85] Korn, W., Teller, P., and Castillo, G. Just how accurate are performance counters? In *Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference* (2002), IEEE, pp. 303–310.

[86] Krishnakumar, R. Kernel korner: kprobes-a kernel debugger. *Linux Journal 2005*, 133 (May 2005), 11.

[87] Lachaize, R., Lepers, B., and Quéma, V. Memprof: A memory profiler for NUMA multicore systems. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012* (2012), pp. 53–64.

[88] Larabel, M., and Tippett, M. Phoronix Test Suite, 2011.

[89] Lawson, N. Side-Channel Attacks on Cryptographic Software. *IEEE Security & Privacy Magazine 7*, 6 (nov 2009), 65–68.

[90] Levon, J., and Elie, P. Oprofile: A system profiler for linux, 2004.

[91] Lin, Y.-B., and Lazowska, E. D. *Reducing the saving overhead for Time Warp parallel simulation.* University of Washington Department of Computer Science and Engineering, 1990.

[92] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium* (2018).

[93] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *Proceedings of the 2015 Symposium on Security and Privacy* (may 2015), IEEE, pp. 605–622.

[94] Liu, X., and Andelfinger, P. Time Warp on the GPU. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation - SIGSIM-PADS '17* (New York, New York, USA, 2017), ACM Press, pp. 109–120.

[95] Luțaș, A., and Luțaș, D. Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction. Tech. rep., Bitdefender, 2019.

[96] Lysenko, M., and D'Souza, R. M. A framework for megascale agent based model simulations on the GPU. *Journal of Artificial Societies and Social Simulation 11*, 4 (2008), 10.

[97] Majo, Z., and Gross, T. R. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of of SYSTOR 2011: The 4th Annual Haifa Experimental Systems Conference, Haifa, Israel, May 30 - June 1, 2011* (2011), P. Ta-Shma, J. Moreira, and L. Shrira, Eds., ACM, p. 12.

[98] Malladi, R. K. Using intel® vtune ™performance analyzer events/ ratios & optimizing applications, 2009.

[99] Malone, C., Zahran, M., and Karri, R. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. In *Proceedings of the 6th Workshop on Scalable Trusted Computing* (New York, New York, USA, 2011), STC, ACM Press, p. 71.

[100] Manual, L. P. Perf_event_open(2), set 2017. Available at: http://man7.org/linux/manpages/man2/perf_event_open.2.html - Accessed: 2017-12-30.

[101] Marathe, J., and Mueller, F. Hardware profile-guided automatic page placement for ccnuma systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006* (2006), pp. 90–99.

[102] Marotta, R., Tiriticco, D., Sanzo, P. D., Pellegrini, A., Ciciani, B., and Quaglia, F. Mutable locks: Combining the best of spin and sleep locks. *Concurrency and computation: practice & experience 32* (2020).

[103] Martin, R., Demme, J., and Sethumadhavan, S. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (jun 2012), ISCA, IEEE, pp. 118–129.

[104] Mavinakayanahalli, A., Panchamukhi, P., Keniston, J., Keshavamurthy, A., and Hiramatsu, M. Probing the guts of kprobes. In *Proceedings of the 2006 Ottawa Linux Symposium* (2006), Ottawa Linux Symposium, pp. 101–115.

[105] McKee, S. A. Reflections on the memory wall. In *Proceedings of the First Conference on Computing Frontiers, 2004, Ischia, Italy, April 14-16, 2004* (2004), S. Vassiliadis, J. Gaudiot, and V. Piuri, Eds., ACM, p. 162.

[106] Merkel, A., Stoess, J., and Bellosa, F. Resource-conscious scheduling for energy efficiency on multicore processors. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010* (2010), pp. 153–166.

[107] Microsystems, S. *UltraSPARC-I - User Manual.* 1997.

[108] Molka, D., Schöne, R., Hackenberg, D., and Nagel, W. E. Detecting memory-boundedness with hardware performance counters. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017* (2017), pp. 27–38.

[109] Moore, G. E. Cramming more components onto integrated circuits. *Proc. IEEE 86*, 1 (1998), 82–85.

[110] Mushtaq, M., Akram, A., Bhatti, M. K., Chaudhry, M., Lapotre, V., and Gogniat, G. NIGHTs-WATCH: A Cache-based Side-channel Intrusion Detector using Hardware Performance Counters. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, jun 2018), ACM, pp. 1–8.

[111] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Understanding measurement perturbation in trace-based data. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA* (2007), pp. 1–6.

[112] Nethercote, N., and Seward, J. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science* (2003), vol. 89, pp. 47–69.

[113] Neville-Neil, G. V. The observer effect. *Commun. ACM 60*, 8 (2017), 29–30.

[114] Nomani, J., and Szefer, J. Predicting Program Phases and Defending Against Side-channel Attacks using Hardware Performance Counters. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy* (New York, New York, USA, 2015), HASP, ACM Press, pp. 1–4.

[115] NOWAK, A., AND BITZES, G. The overhead of profiling using PMU hardware counters, July 2014.

[116] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: The Case of AES. In *Lecture Notes in Computer Science*, D. Pointcheval, Ed., LNCS. 2006, pp. 1–20.

[117] OZCELIK, B., AND YILMAZ, C. Seer: A lightweight online failure prediction approach. *IEEE Transactions on Software Engineering 42* (2016), 26–46.

[118] PALANISWAMY, A. C., AND WILSEY, P. A. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation* (1993), PADS, ACM, pp. 127–134.

[119] PATEL, N., SASAN, A., AND HOMAYOUN, H. Analyzing Hardware Based Malware Detectors. In *Proceedings of the 54th Annual Design Automation Conference* (New York, NY, USA, jun 2017), DAC, ACM, pp. 1–6.

[120] PAYER, M. HexPADS: A Platform to Detect "Stealth" Attacks. In *Engineering Secure Software and Systems*, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds., vol. 9639 of *LNCS*. Springer, Cham, 2016, pp. 138–154.

[121] PELLEGRINI, A. *Techniques for Transparent Parallelization of Discrete Event Simulation Models*. PhD thesis, Sapienza, University of Rome, 2014.

[122] PELLEGRINI, A., AND QUAGLIA, F. The ROme OpTimistic Simulator: A tutorial. In *Proceedings of the Euro-Par 2013: Parallel Processing Workshops*, D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Constan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, Eds., PADABS. LNCS, Springer-Verlag, 2014, pp. 501–512.

[123] PELLEGRINI, A., AND QUAGLIA, F. NUMA time warp. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (New York, NY, USA, June 2015), SIGSIM PADS, ACM, pp. 59–70.

[124] PELLEGRINI, A., VITALI, R., AND QUAGLIA, F. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems 26*, 6 (2015), 1560–1569.

[125] PENG, H., WEI, J., AND GUO, W. Micro-architectural Features for Malware Detection. In *Advanced Computer Architecture*, J. Wu and L. Li, Eds., vol. 626 of *CCIS*. 2016, pp. 48–60.

[126] PETTERSSON, M. Perfctr: Linux performance monitoring counters kernel extension. *URL http://user. it. uu. se/mikpe/linux/perfctr* (2011).

[127] Pierce, C. Detecting spectre and meltdown using hardware performance counters, 2018. https://www.endgame.com/blog/technical-blog/detecting-spectre-and-meltdown-using-hardware-performance-counters - Accessed: 16-08-2018.

[128] Preiss, B. R., Loucks, W. M., and Macintyre, I. D. Effects of the Checkpoint Interval on Time and Space in Time Warp. *ACM Transactions on Modeling and Computer Simulation 4*, 3 (jul 1994), 223–253.

[129] Quaglia, F. Event History Based Sparse State Saving in Time Warp. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation* (1998), IEEE Computer Society, pp. 72–79.

[130] Quaglia, F. A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems 12*, 4 (apr 2001), 346–362.

[131] Quaglia, F., and Santoro, A. Non-Blocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation. *IEEE Transactions on Parallel and Distributed Systems 14*, 6 (2003), 593–610.

[132] Rahman, S., Abu-Ghazaleh, N., and Najjar, W. Pdes-a: Accelerators for parallel discrete event simulation implemented on fpgas. *ACM Trans. Model. Comput. Simul. 29*, 2 (apr 2019).

[133] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the ACM Conference on Computer and Communications Security* (2009), CCS, pp. 199–212.

[134] Rönngren, R., and Ayani, R. Adaptive Checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation* (1994), Society for Computer Simulation, pp. 110–117.

[135] Ronngren, R., Liljenstam, M., Montagnat, J., and Ayani, R. Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation* (1996), PADS, IEEE, pp. 70–77.

[136] Rostedt, S. The x86 NMI iret problem. `https://lwn.net/Articles/484932/`, Mar. 2012. Accessed: 2022-4-14.

[137] Santoro, A., and Quaglia, F. Transparent optimistic synchronization in the high-level architecture via time-management conversion. *ACM Transactions on Modeling and Computer Simulation 22*, 4 (2012), 21:1—-21:26.

[138] Schordan, M., Oppelstrup, T., Jefferson, D. R., Barnes, P. D., and Quinlan, D. Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application. In *Proceedings*

of the 2016 ACM-SIGSIM Conference on Principles of Advanced Discrete Simulation (2016), PADS, ACM Press.

[139] Sharangpani, H., and Arora, H. Itanium processor microarchitecture. *IEEE Micro 20*, 5 (2000), 24–43.

[140] Singh, K., Bhadauria, M., and McKee, S. A. Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architecture News 37*, 2 (2009), 46–55.

[141] Skold, S., and Rönngren, R. Event Sensitive State Saving in Time Warp Parallel Discrete Event Simulation. In *Proceedings of the 1996 Winter Simulation Conference* (1996), Society for Computer Simulation, pp. 653–660.

[142] Soliman, H., and Elmaghraby, A. An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation. *IEEE Transactions on Parallel and Distributed Systems 9*, 10 (1998), 947–951.

[143] Suciu, A., Banescu, S., and Marton, K. Unpredictable random number generator based on hardware performance counters. In *Digital Information Processing and Communications* (2011), Springer Berlin Heidelberg, pp. 123–137.

[144] Sutter, H. The free lunch is over a fundamental turn toward concurrency in software.

[145] Torres, G., and Liu, C. Can Data-Only Exploits be Detected at Runtime Using Hardware Events?: A Case Study of the Heartbleed Vulnerability. In *Proceedings of the 2016 Conference on Hardware and Architectural Support for Security and Privacy* (New York, New York, USA, 2016), HASP, ACM Press, pp. 1–7.

[146] Treibig, J., Hager, G., and Wellein, G. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *39th International Conference on Parallel Processing, ICPP Workshops 2010, San Diego, California, USA, 13-16 September 2010* (2010), pp. 207–216.

[147] Tromer, E., Osvik, D. A., and Shamir, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology 23*, 1 (jan 2010), 37–71.

[148] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. FORESHADOW: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium* (2018).

[149] Van Schaik, S., Giuffrida, C., Bos, H., and Razavi, K. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder than you Think. In *Proceedings of the 27th USENIX Security Symposium* (2018).

[150] Vila, P., Kopf, B., and Morales, J. F. Theory and Practice of Finding Eviction Sets. In *IEEE Symposium on Security and Privacy* (may 2019), SP, IEEE, pp. 39–54.

[151] Wang, X., and Karri, R. Reusing Hardware Performance Counters to Detect and Identify Kernel Control-Flow Modifying Rootkits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35*, 3 (mar 2016), 485–498.

[152] Wang, X., Konstantinou, C., Maniatakos, M., and Karri, R. Con-Firm: Detecting Firmware Modifications in Embedded Systems using Hardware Performance Counters. In *Proceedings of the 2015 International Conference on Computer-Aided Design* (nov 2015), ICCAD, IEEE, pp. 544–551.

[153] Wang, X., Konstantinou, C., Maniatakos, M., Karri, R., Lee, S., Robison, P., Stergiou, P., and Kim, S. Malicious Firmware Detection with Hardware Performance Counters. *IEEE Transactions on Multi-Scale Computing Systems 2*, 3 (jul 2016), 160–173.

[154] Wang, Z., and O'Boyle, M. F. Mapping parallelism to multi-cores: A machine learning based approach. *SIGPLAN Not. 44*, 4 (Feb. 2009), 75–84.

[155] Weaver, V. Linux perf event features and overhead.

[156] Weaver, V. M., Terpstra, D., and Moore, S. Non-determinism and overcount on modern hardware performance counter implementations. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software, Austin, TX, USA, 21-23 April, 2013* (2013), pp. 215–224.

[157] Weisse, O., Bulck, J. V., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Proceedings of the 27th USENIX Security Symposium* (2018).

[158] West, D., and Panesar, K. Automatic Incremental State Saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation* (1996), PADS, IEEE, pp. 78–85.

[159] Winter, M. Data center consolidation: A step towards infrastructure clouds. In *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong, Eds., vol. 5931 of *Lecture Notes in Computer Science*. Springer International Publishing, Berlin Heidelberg, Germany, 2009, pp. 190–199.

[160] Yang, M., Andelfinger, P., Cai, W., and Knoll, A. Evaluation of Conflict Resolution Methods for Agent-Based Simulations on the GPU.

[161] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium* (2014), pp. 719–732.

[162] Yasin, A. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014* (2014), IEEE Computer Society, pp. 35–44.

[163] Yuan, L., Xing, W., Chen, H., and Zang, B. Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks using Performance Counters. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems* (New York, New York, USA, 2011), APSys, ACM Press, p. 1.

[164] Yubin Xia, Yutao Liu, Chen, H., and Zang, B. CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters. In *Proceedings of the 2012 International Conference on Dependable Systems and Networks* (jun 2012), DSN, IEEE, pp. 1–12.

[165] Zaparanuks, D., Jovic, M., and Hauswirth, M. Accuracy of performance counter measurements. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software* (apr 2009), IEEE, pp. 23–32.

[166] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, New York, USA, 2012), CCS, ACM Press, p. 305.

[167] Zhou, B., Gupta, A., Jahanshahi, R., Egele, M., and Joshi, A. Hardware Performance Counters Can Detect Malware. In *Proceedings of the 2018 Asia Conference on Computer and Communications Security* (New York, New York, USA, 2018), ASIACCS, ACM Press, pp. 457–468.

[168] Zhou, H., Wu, X., Shi, W., Yuan, J., and Liang, B. HDROP: Detecting ROP Attacks Using Performance Monitoring Counters. In *Information Security Practice and Experience*, X. Huang and J. Zhou, Eds., vol. 8434 of *LNCS*. 2014, pp. 172–186.