Received 24 January 2023 | Revised 12 May 2023 | Accepted 6 June 2023

DOI: 10.1002/spe.3236

RESEARCH ARTICLE

Strategies and Software Support for the Management of Hardware Performance Counters

Stefano Carnà[1] | Romolo Marotta[2,3] | Alessandro Pellegrini[2] | Francesco Quaglia[2]

[1]DIAG, Sapienza, University of Rome, Italy
[2]DICII, University of Rome “Tor Vergata”, Italy
[3]Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing, Italy

Correspondence
Alessandro Pellegrini, DICII, University of Rome Tor Vergata, Rome, Italy.
Email: a.pellegrini@ing.uniroma2.it

Summary

Hardware Performance Counters (HPCs) are facilities offered by most off-the-shelf CPU architectures. They are a vital support to post-mortem performance profiling and are exploited by standard tools such as Linux or Intel V-Tune. Nevertheless, an increasing number of application domains (e.g., simulation, task-based high-performance computing, or cybersecurity) are exploiting them to perform different activities, such as self-tuning, autonomic optimization, and/or system inspection. This repurposing of HPCs can be difficult, e.g., because of the overhead for extracting relevant information. This overhead might render any online or self-tuning activity ineffective. This article discusses various practical strategies to exploit HPCs beyond post-mortem profiling, suitable for different application contexts. The presented strategies are accompanied by a general primer on HPCs usage on Linux. We also provide reference x86 (both Intel and AMD) implementations targeting the Linux kernel, upon which we present an experimental assessment of the viability of our proposals.

KEYWORDS:
Hardware Performance Counters, Self-Tuning, Profiling, Autonomic Computing.

1 | INTRODUCTION

Most families of CPUs currently include specialized components within their architecture, collectively named Hardware Performance Counters (HPCs) or Performance Monitor Counters (PMCs), which can be used to gather information about what is going on at the hardware level during code execution. While HPCs 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. In fact, they have been used not only to profile the execution of software in terms of actual hardware reactiveness and achievable performance[1], or for relating software execution to power and energy consumption[2,3], but also to support security[4] or specific functional tasks[5] thus offloading them from software modules.

At their simplest, HPCs come in the form of a set of programmable hardware registers directly connected to the circuitry of the CPU and controlled by a Performance Monitoring Unit (PMU). Once they are programmed to track a specific hardware-level event (e.g., a cache hit at a particular cache level), their value is automatically and transparently increased every time such events occur through dedicated data paths in the chip. This makes them particularly efficient. Due to their importance, many additional extensions (e.g., the Intel Precise Event-Based Sampling—PEBS[6], which allows collecting also memory addresses involved in application access) have been introduced over the years.

Profiling has been for sure one of the core motivations for the evolution of HPCs, and well-established profilers have extensively used them (e.g., Linux `perf` or Intel V-Tune). Also, recent research lines have highlighted the opportunity to rely on HPCs' innovations 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[7,8]. In this direction, HPCs have been used in a myriad of applications, such as dynamic software profiling[9], CPU power modelling[2], children privacy protection[10], failure prediction in computing systems[11], random number generators[12], or for the generation of test programs via fuzzing to enhance code coverage[13] just to mention a few diverse applications.

These solutions share the need for efficient management and analysis of the generated samples, which is the activity associated with the most considerable overhead. Indeed, while HPCs are fast and non-intrusive at generating samples, managing these samples to create higher-level information to be exploited online is costly.

The overhead introduced in the system to collect and manage HPC samples is thus relevant for many modern applications. This overhead is exacerbated if an application requires the collection of these samples at a higher frequency—this is the case of HPC exploitation for supporting incremental checkpointing via the HPC-based identification of memory areas accessed in write mode[5].

Beyond pure overhead aspects for the collection and management of HPC samples, we also need to consider that i) HPCs, in general, do not differentiate between user space and kernel space, therefore if an application wants to filter out data related to execution in kernel mode, this must be explicitly done; ii) HPCs do not differentiate between processes; therefore the samples should be manually filtered; iii) many sources in the system might induce some non-deterministic behaviour[14], which should be explicitly taken into consideration; iv) performance counters may overcount certain events on some processors[14].

Some of these aspects—like points i) and ii)—are explicitly dealt with by standard tools exploiting HPCs, such as Linux `perf` or Intel V-Tune. Nevertheless, such tools only offer post-mortem capabilities to exploit HPC data. Furthermore, as we also show in our experimental analysis, they can lose PMU samples under the high frequency of their production, which can be undesirable in specific application contexts, like for example PMU exploitation for security[4]. Hence, how to cope with all the aforementioned aspects in the context of online exploitation of HPCs is still an ongoing research activity.

In this article, we propose and discuss several approaches which can be used to program, control, and read data from HPCs. Our approaches try to solve some or all of the limitations mentioned above, thus serving as building blocks for online operations. 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[1] with an extensive assessment, which shows the benefits and limitations of each solution.

The interactions between the software layer and HPCs has some technical difference among the various CPU architectures. We tailor our solutions for x86 architectures, encompassing both Intel and AMD chipsets—we explicitly focus on hardware supports from different CPU generations. Nevertheless, our approaches can be easily ported to other architectures. Our reference implementations target the Linux kernel. Furthermore, rather than requiring kernel recompilation and re-installation, we target an implementation based on a Linux kernel Module (LKM) just for ease of use—it enables loading/unloading the required kernel-level software at runtime.

The remainder of this article is structured as follows. In Section 2, we provide the background on HPC facilities offered by modern off-the-shelf CPUs. In Section 3, we discuss related work. The various interaction schemes with HPCs are presented in Section 4. Our experimental assessment is provided in Section 5.

2 | BACKGROUND

HPCs enable an *event-based sampling*, 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 that can occur in architectural elements because of 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.

[1] Available at https://doi.org/10.5281/zenodo.6621964

In the literature, this extremely low-level information is often referred to as *micro-architectural events*. The benefit of relying on micro-architectural events is that their support is highly optimised and does not slow down the speed of the CPU, which is highly desirable. At the same time, micro-architectural events are directly linked to design choices and may vary among different manufacturer products, accounting for innovative features shipped with new design generations. Consequently, there is no guarantee that a micro-architectural event can be monitored with different processor models or 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 some events, called *architectural events*, that are deemed common to different processor models. Relying on architectural events increases the portability of solutions based on these supports, although they are typically reduced in number.

Modern CPUs support hundreds of micro-architectural events, but only a handful of HPCs are available (at the time of this writing, off-the-shelf CPUs offer between 4 and 8 HPCs per core). Among the rich set of events that can be monitored, it is possible to identify the following classes:

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), expected to be incremented at each clock cycle.
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. *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 information on branch instructions, such as the number of branch mispredictions or retired branch instructions.

On typical off-the-shelf architectures, each HPC can be configured to 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 typical way to access these counters is through a read/write on model-specific registers (MSRs) exposed by the underlying architecture. 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`[15,16] instructions (which are privileged instructions), and it is also possible to read the HPC from userspace via the `rdpmc` instruction[15].

An HPC can be configured to work in two different modes: *counting* and *sampling* modes. The counting mode makes the counter register simply accumulate 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 investigate the context that generated the hardware event.

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. This aspect is exacerbated 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 orderings associated with micro-operation execution and instruction retirement, the actual sampling notification may be late compared to its generation point. This delay is called *skid*. Its presence is the reason behind additional capabilities embedded within PMUs, which we describe in the following sections.

2.1 | Intel's Precise Event-Based Sampling (PEBS)

PEBS[15,6] is an Intel extension to traditional HPCs to increase their precision. It is an event-based sampling 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

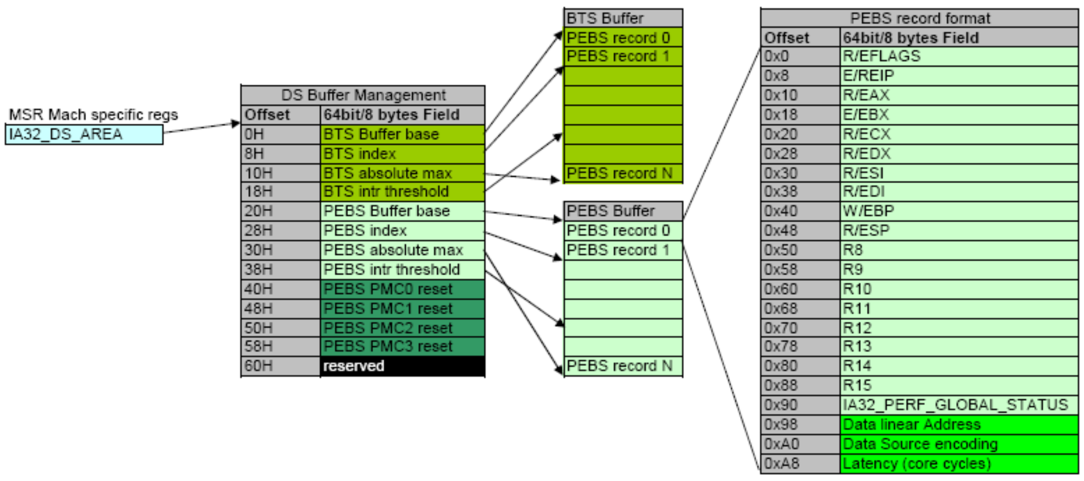


FIGURE 1 Organization and relation of the data structures used in the context of PEBS.

top HPCs, it does not take complete control over them: it is, therefore, possible to exploit both PEBS and traditional HPCs simultaneously.

PEBS cannot be applied to all the events supported by PMUs, yet it provides a new hardware-based mechanism that automatically saves the processor context when the counter 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 PMU 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 an architectural event monitored using the PEBS support is triggered, a data sample is produced to snapshot the whole CPU state. In the PEBS buffer, the next record to be filled with a newly-generated sample is identified thanks to the *PEBS index*. When the PEBS index reaches the *PEBS threshold*, which can be configured at setup time, a hardware interrupt is used to notify that the buffer is almost full, and a read operation should be carried out as soon as possible to avoid losing samples. The overall organisation and the relation between the data structures used to support PEBS execution are depicted in Figure 1.

2.2 | AMD's Instruction-Based Sampling (IBS)

IBS[16,17] is another precise support that adopts a different methodology for generating data that, in some ways, can be considered as a variation of the time-based sampling (TBS). The yardstick of this sampling technique is the instruction[2]: 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 a threshold is reached. The magic lies in the hardware-firmware cooperation that is in charge of collecting the CPU state combined with additional information at the precise moment the culprit instruction is fetched[3] from the CPU frontend and following its execution through the entire pipeline until its retirement. All the events generated by the instruction computation are recorded into dedicated MSRs, realising a complete execution snapshot. IBS defines an orthogonal solution to PEBS, providing a detailed information set that aims for a comprehensive and complete analysis rather than studying more specific dynamics.

[2]Actually the user can define if the counting process has to be based on either executed micro-operations or elapsed clock cycles.
[3]IBS is split into the *fetch* and the *execution* units that provide information concerning the processor frontend and backend, respectively.

3 | RELATED WORK

Since their initial implementations, PMUs have been largely investigated in the literature, providing a full spectrum of their capabilities. At the same time, despite their implementation by means of dedicated circuitry, the overhead introduced for their usage is not always negligible[18,14] and can sensibly vary according to the profiling mode, the frequency period, and the techniques adopted to collect and read the gathered data. This reinforces the well-known accuracy/overhead tradeoff, which at high rates might slow down the entire system's execution (making the exploitation of PMUs too invasive for online tasks) and also introduces side-effects such as profile data perturbation, making the measurements possibly unreliable[19,6].

HPCs lack a standard interface. The policies to interact with this support differ across processor vendors and can require different actions even among models of the same vendor. Consequently, most hardware-enhanced software analysis programs rely on external libraries or tools to nimbly integrate PMU functionality and extend solution portability. OProfile[20], Perfctr[21], Perfmon2[22] or PAPI[23] are some of the most used interfaces exploited by 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, its goal is only related to simplifying the access to PMUs on various architectures, while the task of interpreting the generated information is still demanded from the user. AMD code analyst[24], Intel VTune[25], HPCTool[26] and LikWid[27] are advanced tools which do not export a PMU interface, but provide a full-fledged set of configuration and interpretation functionalities. Indeed, they provide a useful UI assisting non-experts during session configuration and offer a built-in analysis tool that pinpoints critical problems in the analysed executions to carry out complex tasks such as performance optimisation.

Nonetheless, they do not provide any means for external software interaction, limiting their activities to local (though very precise) application profiling, which cannot be exploited in online tuning actions. The most versatile framework in the Linux system is `perf_events`[28]. It is directly integrated into 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 elements to support the realisation of self-tuning software components, Weaver[14] demonstrated how its evolution path introduced system performance degradation in favour of more advanced features integration. Generally, profilers do not care about this drawback because they just gather data to perform post-mortem analysis. Even so, this represents a fundamental limitation for techniques acting in an online fashion, which requires the identification of the suitable trade-off between the generated data and the polling frequency of such information without overloading the system and perturbing the original program flow with the profiling activity[29].

Despite all the highlighted problems, hardware monitoring support has been used to address disparate research challenges. HPCs have been employed to improve *energy efficiency*[30] of systems. In this context, Singh et al.[31] proposed a new methodology to estimate real-time power consumption and devise a power-aware system scheduler. Wang and Boyle[32] showed that hardware instrumentation could drive compilers to achieve optimal mapping of program parallelism on the underlying architecture by performing automatic profiling runs. The works in[33,34] have focused on the properties of HPCs that are needed for accurate power/energy modelling. These works also studied how raw HPC data collected using existing Linux solutions satisfy these properties. The work in[35] presents a study on how to combine HPC data with utilization variables for the accurate modeling of power/energy usage.

Various methods have been proposed for correcting sampled HPC values, which can be classified into two main groups. The first group interpolates between two sampled events using linear or piece-wise interpolation to artificially fill data gaps in collected samples (e.g.,[28]). However, these interpolation methods may not provide accurate imputations[36] despite being able to run in real-time. The second group of methods correct measurements by removing outlier values instead of adding new interpolated values. These methods require the entire trace of an application before providing corrections and cannot be run in real time. For example, Lv et al.[37] use the Gumbel test for outlier detection, and Neill et al.[38] use fork-join aware agglomerative clustering to remove outlier points. These methods are unsuitable for dynamic control situations requiring online HPC correction. Furthermore, the core statistical technique these variance methods use is limited in effectiveness. BayesPerf[39] uses statistical relationships between events to correct measurements accurately. When dealing with well-documented processors, these relationships can be known in advance, allowing the entire correction algorithm to be executed without the need to pre-collect data. BayesPerf allows corrected HPCs to be accessed with nearly native latency, making them suitable for dynamic control processes. Differently from these proposals, we focus on developing a technique that effectively enables using HPC values for non-functional operations.

Many supports have been diagnosed with several elusive problems[40] and anomaly detection[41] 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 adopted PMUs to retrieve low-overhead and accurate information on allocator choices and memory access path[42]. 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 optimisation in a subsequent phase[43]—or a pure online logic exploiting specific experimental measurements for parameters tuning, which indeed does not represent a flexible solution over general application domains[44]. Molka et al.[45] studied the memory footprint of an application detecting its memory-boundedness, also highlighting the complexity to choose the right metrics[14]. In the context of speculative High-Performance Simulation, HPCs have been used to trace memory accesses to build incremental logs of application states to support low-overhead memory checkpoint/restore activities[5].

Hardware introspection capabilities provided by HPCs are suitable to address multiple security challenges[46,47], ranging from the identification of Return-Oriented Programming (ROP) attacks[48] to the detection of several exploits of side-channel vulnerabilities[49,50]. Furthermore, the design of countermeasures for new *transient execution vulnerabilities*[51] present in most processors required new strategies featured by hardware instrumentation[52,53] to observe the stealth activities performed by attackers, also in a system-wide fashion[4]. All these works highlight the relevance of PMUs as a support for the online profiling of applications. Nevertheless, carrying out this kind of activity might incur severe performance penalties. Tackling these issues, while still providing a general-purpose tool for accessing HPCs in an easy to configure and efficient manner, is the goal of this article.

4 | EFFICIENT ONLINE GATHERING OF HPC DATA

In this section, we provide a comprehensive panoramic of the various techniques that could enable effective exploitation of PMUs for online operations (e.g., inline optimisation), possibly covering all the research lines mentioned in Section 3. Our techniques are discussed as a way to realise a *profiling agent* that works partially in kernel mode and partially in user mode. The kernel-mode agent can be implemented as a Loadable Kernel Module (LKM) or a patch to the kernel, depending on its inner workings. We stress its implementation via LKM just for ease of use (it can be installed with no kernel recompilation and installation).

4.1 | Efficient Collection of PMU Data

In common tools, like the well-known `perf_events` in Linux[28], PMUs are configured to generate *performance monitor interrupts* (PMIs) on the Non-Maskable Interrupt (NMI) line. This interrupt 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. On the x86 architecture, each core has a single non-maskable interrupt line[4]. 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.

Directly modifying the interrupt management strategy—rather than relying on NMI via the installation of an additional NMI handler—can be an effective intervention to achieve a more timely system response. In particular, in our proposal, 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 before and after the actual management routine is activated.

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:

1) it checks if we are coming from a user- or a kernel-mode and, if needed, swaps the `GS` segment;
2) it creates a complete registers snapshot, swaps to the kernel-level page table (this is required for hardware-level security bugs like Meltdown[54], and is supported by the Page-Table-Isolation service[55], namely, PTI) and switches to the dedicated IRQ stack;

[4]All NMIs go into interrupt line 2.

3) it calls the `do_IRQ` function, which sets some system flags and executes the actual handler;
4) it leaves the IRQ stack;
5) it calls the kernel function in charge of managing the interrupt;
6) it restores the registers snapshot and executes `swapgs` and the swap to the user-level page table if required;
7) it returns from interrupt.

On x86 architectures, the number of available *interrupt descriptor table* (IDT) vectors is limited to 256, each associated with a dedicated management routine, even if that specific interrupt is not actually used by the operating system—it is a spurious one. This is done in order to allow the kernel to actually access the software needed to respond to an interrupt (or a trap) under any circumstance.

In our solution, the exploitation of a specific entry of the IDT to manage PMU interrupts has been organized without the need for a new entry installation on the IDT. In fact, this solution would require the inclusion of the IRQ routine at the level of the kernel entry portion, already visible in user-level page tables. This would require, in turn, recompiling and reinstalling the kernel. Instead, to avoid this problem and to exclusively rely on an LKM, we identify one IDT entry hosting the spurious interrupt IRQ routine, and then we simply binary patch the call carried out by that routine in point 5) of the above task list (directed to the platform-independent manager of the spurious interrupt) to our own manager embedded within the LKM. Given that the kernel-level page table gets opened by the IRQ routine (see point 2) in the above task list) before the call is executed, our handler is correctly visible and usable by the CPU, even if it stands in the memory area used for an LKM. Clearly, this solution still works if the kernel is booted with the no-page-table-isolation option since, in this case, the IRQ routine we binary patch still offers the call instruction we have exploited. Also, in order to correctly identify that instruction, we exploit the "kallsyms" service offered by the Linux kernel—particularly via the `kallsyms_lookup_name()` function—to discover where the original handler of the spurious interrupt stands in the memory address space. This is done by simply passing the name of the `smp_spurious_interrupt` kernel handler to the `kallsyms_lookup_name()` service. This enables us to perform pattern matching of the call-instruction displacement, so as to locate the correct call to be binary patched. This solution still works with kernel-level randomization of the address space, since the kallsyms service is offered at runtime.

Thanks to this approach, the PMI routine has direct access to the code managing HPC samples because such an IDT entry is not shared with other routines/handlers. Consequently, we save the cost of polling multiple handles, which is instead spent when using NMI. Clearly, the higher the frequency of PMI events, the larger the benefits of our approach and the overhead reduction. This aspect is deeply discussed in our experimental study in Section 5.

4.2 | Discriminating the Profiling Domain

PMUs work at the architectural level, thus ignoring operating system dynamics such as context switches. In a time-sharing system, discriminating what application was running when a PMU captured a particular event needs therefore specific support at the software level. Therefore, *thread discrimination* is an essential feature for the association of PMU data to the activities carried out on behalf of any application.

Furthermore, another relevant aspect to consider is *mode discrimination*, namely the capability to filter events depending on the privilege level when they occur. While x86 architectures offer four different privilege levels, only two are used in practice by standard off-the-shelf operating systems. PMUs abide by this practice and offer specific bits in control MSRs to select the desired mode discrimination. Regrettably, this Current-Privilege-Level (CPL) filter is not reliable on many architectures[56].

This section presents different software-level solutions that we devise to offer both thread and mode discrimination, with reduced overhead. This makes them suitable for online PMU-exploitation purposes.

4.2.1 | Per-thread Profiling

It is possible to implement per-thread filtering of the generated PMU data provided that: i) we can associate a memory area to store the trace of generated samples with the current thread; ii) we can intercept a context switch to determine what memory area needs to be used for the newly CPU-dispatched thread; iii) we can let the system know the pool of threads that the profiling agent should actively manage.

To cope with these aspects, we propose an explicit "registering" procedure, which allows the profiling 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.

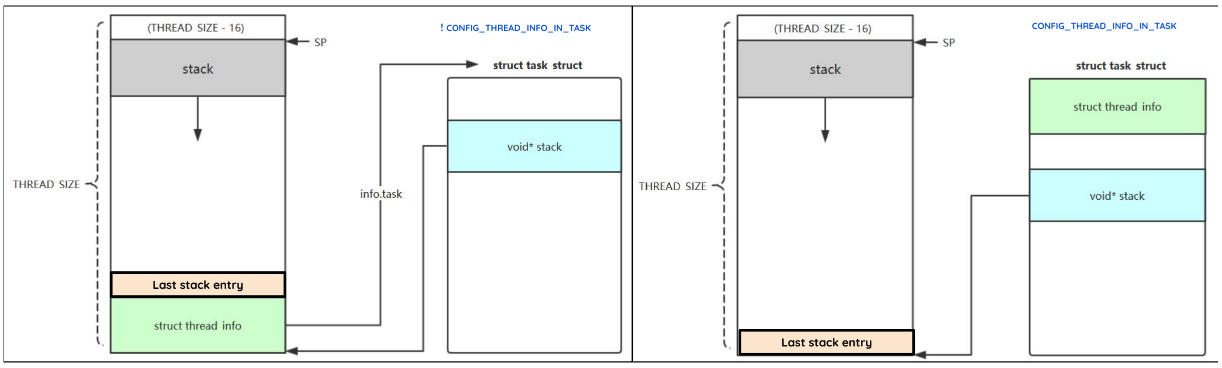


FIGURE 2 Flagging a Thread on its Kernel Stack.

This registering phase allocates all the required memory needed by our profiling agent for each registered thread (e.g., memory buffers for storing the sample trace). Since we are interested in maintaining the overhead as low as possible, retrieving the location of sample buffers for a given thread must be a very efficient activity, especially considering that such a step could be performed at least once for each context switch. Hence, the profiling agent should leverage some efficient data structure to perform thread lookups.

For this purpose, we could extend the `task_struct` (a process control block, aka PCB, associated with the thread) with a dedicated flag to keep the monitoring state and a pointer to the memory area used to host the sample trace. While this solution appears immediate, it is not viable if the monitoring agent is implemented purely via an LKM, since it requires recompiling the kernel. Consequently, we explored three different approaches for which we provide performance data in Section 5.

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.

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 `task_struct` that we can use to flag a thread as under monitoring. We therefore take advantage of the possibility of registering 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, e.g. references to memory buffers for storing the currently generated sample trace—we will discuss this aspect in more detail in Section 4.3. Listing 1 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.

The third solution is based on storing a monitoring flag directly on the kernel stack of the profiled thread. Figure 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 associated with the thread, which can be accessed in constant time when that thread 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 `thread_info` data structure at the end of the stack area, as shown in Figure 2. Therefore, care must be taken to determine the proper placement of the `thread_info` data structure, not to overwrite its content—this is a check that can be performed at the LKM compile-time anyhow.

Listing 1 Example of perf_event Creation, Release and Query Functions.

```c
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 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. One 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 hyper-thread on board of the processor can leverage its own set. Nevertheless, both relying on a CRC check or a debug register introduces additional overhead to the simple task of determining whether a thread should be profiled or not.

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 switches a thread off the

[5]This problem is one reason that led kernel developers to map the kernel stack from physical onto virtual memory and to move the `thread_info` data structure out of the stack, as also shown in Figure 2.

CPU, the PMU data of the thread must be updated to allow resetting the profiling trace. Actually, 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 a redirection of 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 2, we present our approach for 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`.

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` should not be called anymore upon a 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 2 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 threads. 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, since reconfiguring the involved MSRs would not bring any gain but only a cost. A PMU's configuration transition occurs only when the switch between two threads bumps into a switch-off/on of the profiling activity. To this end, we have introduced the per-CPU `pcpu_pmcs_active` variable 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 exception handler is activated to implement the logic associated with the management of the kprobe.

The exception handler knows the installed probe and carries out the additional logic before or after instrumenting the code instruction. 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 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.

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

[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.

Listing 2 Context Switch Callback based on Tracepoints.

```c
struct hook {
        char *name;
        void *func;
        struct tracepoint *tp;
};

static void ctx_hook_func(void *data, bool preempt, struct task_struct *prev,
                  struct task_struct *next))
{
    if (this_cpu_read(pcpu_pmcs_active) && !is_profiled(next))
                // Do something with prev's profiling data (see Section 4.3)
                disable_pmc_on_this_cpu();
        else if (!this_cpu_read(pcpu_pmcs_active) && is_profiled(next))
                enable_pmc_on_this_cpu();
}

static struct hook ctx_hook = {"sched_switch", ctx_hook_func, NULL};

static void lookup_tracepoint(struct tracepoint *tp, void *hook_ptr)
{
        struct hook *hook = (struct hook *)hook_ptr;

        if (strcmp(hook->name, tp->name) == 0)
                hook->tp = tp;
}

int register_ctx_hook(char *name, void *func)
{
        /* Fill the hook's tracepoint */
        for_each_kernel_tracepoint(lookup_tracepoint, hook);

        if (!hook->tp)
                goto err;

        if (tracepoint_probe_register(hook->tp, hook->func, NULL))
                goto err;

        return 0;
err:
        return -ENXIO;
}

void unregister_ctx_hook(enum hook_type type, void *func)
{
        tracepoint_probe_unregister(hook->tp, hook->func, NULL);
        tracepoint_synchronize_unregister();
}
```

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 `current`. We provide in Listing 3 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

Listing 3 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;
}
```

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 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 4 shows the reference implementation of the callbacks when relying on both techniques.

4.2.2 | Software-based CPL Filtering

One desired feature demanded from a monitoring agent is being able to discriminate whether some monitored events are related to a thread's execution in either user mode or kernel mode. More formally, we can define the function $P(x, y, z, t)$ that returns the occurrences counted by the HPC x for an event y under the execution mode z during a time period t. We can consider the monitoring precise if we can assert that:

$$P(0, \alpha, user, \delta_{t0}) + P(1, \alpha, kernel, \delta_{t0}) = P(2, \alpha, user + kernel, \delta_{t0}) \tag{1}$$

Listing 4 Intercepting Thread Creation/Termination Events.

```c
// Static Tracepoints
void fork_hook_func(void *data, struct task_struct *parent, struct task_struct *child)
{
        if (is_profiled(parent))
                pid_register(child);
}

void exit_hook_func(void *data, struct task_struct *p)
{
        pid_unregister(p->pid);
}

// Dynamic Kprobes
int exit_pre_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        pid_unregister(current->pid);
        return 0;
}

int fork_ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        unsigned long retval = 0;

        if (is_profiled(current)) {
                retval = regs_return_value(regs);
                if (is_syscall_success(regs))
                        /* Fork return value is the child's pid */
                        pid_register(retval);
        }
        return 0;
}
```

In other words, we have precise monitoring if, relying on three different HPCs—one tracking user-mode events, one tracking kernel-mode events, and one tracking both—no event is lost, and we can create some distinction (as accurate as possible) between the events occurred in user and kernel mode.

Intel's implementation of HPCs allows a CPL filtering bit-mask to define the execution mode in which events should be observed (user mode, kernel mode, or both). As discussed, this capability is unreliable due to a functional problem in the firmware[56] in many architecture generations—performance counters may completely miss some events when CPL filtering is enabled. It means that, in the scenario mentioned above, the following relation holds:

$$P(0, \alpha, user, \delta_{t0}) + P(1, \alpha, kernel, \delta_{t0}) \leq P(2, \alpha, user + kernel, \delta_{t0}) \tag{2}$$

We discuss in this section a strategy to implement software-based filtering that allows implementing the correct behaviour described by Equation (1) also on architectures that cannot provide this feature at the hardware level. Our approach is based on letting PMUs run without any hardware filtering, i.e. making them collect profiling information related to both kernel and user mode execution. We extend per-thread data as discussed in Section 4.2.1 by introducing two extra members to hold PMU snapshots in order to separate values while running either in user or kernel mode. Then, the fundamental aspect is to identify thread transitions from user mode to kernel mode and vice versa. Suppose we can detect these events on a per-CPU basis. In that case, we can update the extra members with the data collected by the PMUs in a specific time window associated with either execution mode, as shown in Figure 3. We note that the approach discussed in Section 4.2.1 is perfectly compatible with software-based CPL filtering because intercepting a context switch is an additional point in which it is possible to update per-thread/per-mode PMU data.

As mentioned, we are interested in intercepting any transition to kernel mode. In particular, the events that we have to detect are the execution of an interrupt trampoline, of the system call handler, or an exception routine. Every time a mode switch is detected, we store an HPC snapshot in a dedicated per-CPU variable. Once returning to userspace, we use the current HPC value to compute the difference from the previous snapshot. This mechanism allows determining the number of events executed in kernel mode. Since PMUs are configured to track events both in user and kernel mode, and we compute the total number of events generated in kernel mode through subsequent snapshots, the number of events executed in user mode can be computed by simple subtraction.

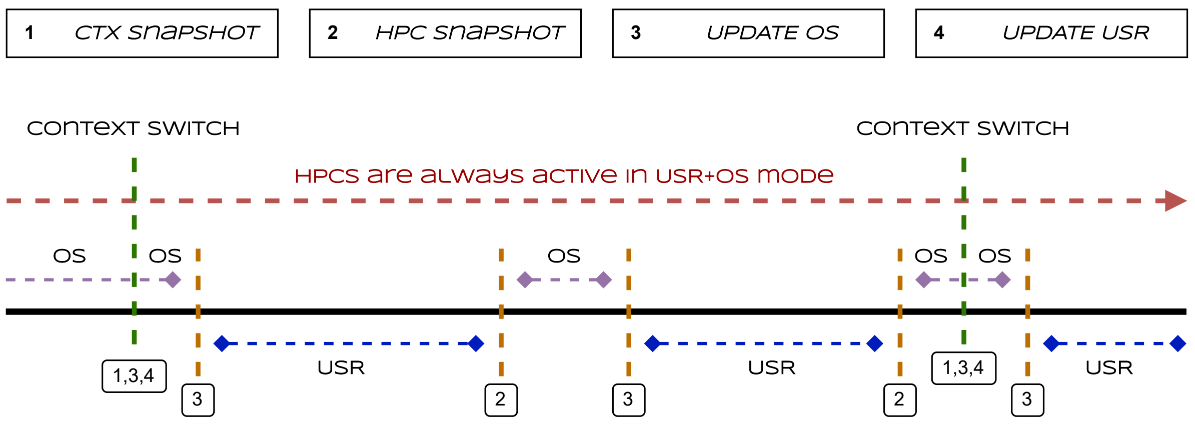


FIGURE 3 Software-based HPC's CPL Filtering Schema.

It is noteworthy that callback events generated to track mode switches might have an actual (minimal) delay compared to the precise moment the switch occurs because some kernel-mode instructions might be executed before reaching the tracepoint or triggering the kprobe. Therefore, our approach could associate with user-mode execution a minimal number of events that actually belong to kernel mode. Nevertheless, we emphasise that our approach ensures that Equation (1) holds in any case, therefore providing a result more reliable for many applications (see, e.g., Weaver et al.[14]) than the hardware-based one.

4.3 | Data Buffering

An additional problem that a monitoring agent has to face to support online PMU data exploitation 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. These are well-known problems, exacerbated by the possibly huge amount of data collected at runtime from PMUs.

In this context, memory management is essential to consider. Indeed, data generated while running PMI code must be retained in some buffer in memory. Consequently, it is required that there is always some free buffer available to avoid sample losses.

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.

Since proposing an allocation scheme that allows solving the above-mentioned issues is beyond the scope of this article, we focused on optimizing the re-usage of a given pre-allocated memory buffer. This allows us to reduce the likelihood of interactions with any memory allocator. Additionally, we also cope with the edge case in which no additional memory can be acquired, a situation that can by overwriting older samples.

In particular, we propose a mostly non-blocking Single-Enqueuer/Multiple-Dequeuer circular queue that allows minimizing the latency required to store a sample and to favour memory reusage without complex synchronization mechanisms (e.g., locking and garbage collectors).

Such a data structure, discussed in Section 4.3.1, can be rearranged to support system-wide, per-core and per-thread collection of samples, discussed respectively in Sections 4.3.2, 4.3.3, and 4.3.4.

4.3.1 | Single Enqueuer Multiple Dequeuer Circular Queue

We propose a candidate data structure that can be used to reduce synchronisation costs and to favour memory reusage, namely a mostly non-blocking[57] Single-Writer/Multiple-Readers circular queue. In this data structure, writes have higher priority over reads, this is because we account for the scenario in which data generation takes place within the PMI, whose duration should be kept as short 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

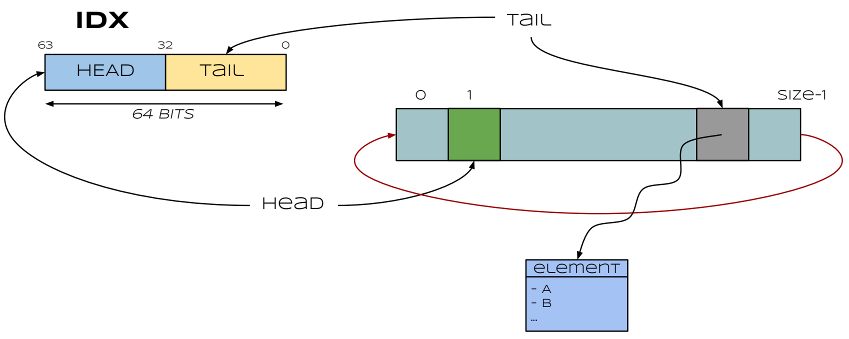


FIGURE 4 General Organisation of the Non-Blocking Circular Queue.

rely on atomic `fetch_and_add` (FAA) and `compare_and_swap` (CAS) instructions, which 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, and 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 4 depicts the organisation of our 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) 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 5:

- EMPTY: head and tail are equal. 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 former will overwrite the oldest element.
- NORMAL: head and tail are different[9], and both operations can be called.

We report in Algorithm 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[10], which is achieved by relying on a replacement strategy of the samples in case the buffer is full. This strategy makes readers wait for write completion (for this reason, the circular queue is mostly non-blocking from a theoretical perspective). 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 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 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.

[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.
[9]The logical value of head can be greater or smaller than the tail, depending on the respective cycles.
[10]A wait-free method is guaranteed to terminate in a finite number of steps.

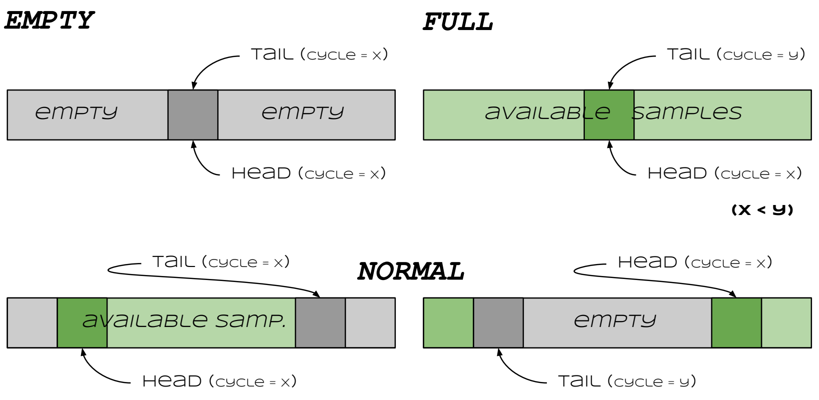


FIGURE 5 Possible Queue States.

Algorithm 1 Enqueue Operation (Single Writer)

```
1:  procedure INSERT( )
2:      $midx \leftarrow$ 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 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. 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—under reasonable scenarios, this condition should occur infrequently. 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 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.

4.3.2 | 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.

Algorithm 2 Dequeue Operation (Multiple Readers)

```
1: procedure REMOVE( )
2:     loop
3:         $midx \leftarrow$ IDX
4:         while HEAD($midx$) + SIZE < TAIL($midx$) do
5:             $midx \leftarrow$ IDX
6:         end while
7:         if HEAD($midx$) = TAIL($midx$) then
8:             return empty
9:         end if
10:        $elem \leftarrow$ READ(HEAD($midx$))          ▷ 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
```

Moreover, accesses from the monitoring agent to deliver performance data to userspace must be synchronised to avoid incorrect reads while write operations occur. Synchronisation likely becomes the performance bottleneck given the high concurrency we expect to generate PMU data.

4.3.3 | Per-core Buffering

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 core is intrinsically responsible for its dedicated PMUs, which provide data related to the events happening on that core [11]. This buffering strategy can be leveraged in scenarios where the effective usage of hardware resources can drive thread placement on a per-core basis[58].

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 circular 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.

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 PMU data exploitation goal, the overhead of this strategy can be too high.

4.3.4 | Per-thread Buffering

A different buffering strategy is related to keeping all performance data separated per thread. As discussed in Section 4.2.1, it is possible to let the monitoring agent associate PMU data to a specific thread. Keeping the 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[59] or to improve memory access latency in NUMA systems[60].

[11] 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 (the hyper-thread.

Keeping per-thread data can be done by relying on multiple strategies. Clearly, the circular queue presented in Section 4.3.2 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 4.2.1, 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 this case, we have to keep sampled data in the dedicated queue. Any data structure used for this purpose can be augmented with a pointer to locate the per-thread queue in memory. 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 4.2.1. As mentioned, its integration in the kernel is such that some data structures already keep members used to support its execution. Referring again to Listing 1, we have shown 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`, allowing us to store custom data inside the event.

4.4 | 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 would benefit from separating the PMU data collection part from its usage logic. In particular, it may be desirable to run the latter in user space, especially for maintainability.

In this scenario, it is fundamental to allow userspace applications to read PMU samples. While it is theoretically possible to rely on the `rdpmc` instruction to read values from HPCs, relying on this instruction can be of little help. Indeed, as we have discussed, reading raw data from HPCs cannot allow for, e.g., discriminating between threads or execution modes. This aspect has been coped with by employing multiple layers of abstraction at the kernel level, which eventually populate data structures that should be read from userspace. Therefore, it is fundamental to bridge the approaches discussed so far with data accesses from user space, which is a more complex task than executing the simple `rdpmc` instruction.

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 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[12].

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 code, which can be undesirable due to higher complexity. Moreover, when dealing with 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 explicitly interact with a linear file 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.

Therefore, while we suggest anyhow to rely on the `proc` file system, we highlight a different strategy, showcased in Listing 5. First, we install a set of pseudo-files to read all possible combinations of data buffers 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 `seq_operations` data structure contains the reference to the `seq_file` operations: `start`, `next`, `show`, and `stop`. Those functions implement the core of the `seq_file` iterator-like navigation proper of the Linux kernel. As we will show experimentally in Section 5.3, this approach is slightly more costly than the `mmap`-based solution due to the various layers that compose the VFS.

[12]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.

Listing 5 Allowing User Space Applications to Obtain Profiling Data.

```
static struct proc_dir_entry *proc_entry;

static void *__breader_seq_next(loff_t *pos)
{
        if (*pos >= DATA_BUFFER_LEN)
                return NULL;
        return &data_buffer[*pos];
}

static void *breader_seq_start(struct seq_file *m, loff_t *pos)
{
        return __breader_seq_next(pos);
}

static void *breader_seq_next(struct seq_file *m, void *v, loff_t *pos)
{
        (*pos)++;
        return __breader_seq_next(pos);
}

static int breader_seq_show(struct seq_file *m, void *v)
{
        uint pmc;
        u64 *pmcs_data = v;

        for_each_pmc(pmc, 8)
                seq_printf(m, ",%llx", pmcs_data[pmc]);
        return 0;
}

static void breader_seq_stop(struct seq_file *m, void *v)
{
        seq_puts(m, "\n"); /* Nothing to free */
}

static struct seq_operations breader_seq_ops = {
        .start = breader_seq_start,
        .next = breader_seq_next,
        .stop = breader_seq_stop,
        .show = breader_seq_show
};

static int breader_open(struct inode *inode, struct file *filp)
{
        return seq_open(filp, &breader_seq_ops);
}

static const struct file_operations breader_fops = {
        .open                = breader_open,
        .read                = seq_read,
        .llseek                = seq_lseek,
        .release        = seq_release
};

int init_seq_file(void)
{
        proc_entry = proc_create("breader_seq", 0444, NULL, &breader_fops);

        if (!proc_entry)
                return -ENODEV;
        return 0;
}

void fini_seq_file(void)
{
        proc_remove(proc_entry);
}
```

5 | EXPERIMENTAL ASSESSMENT

This section presents an experimental assessment of the performance implications of the different techniques discussed in this article. 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[61], 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 reads 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*: performs 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 Section 4. It 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.

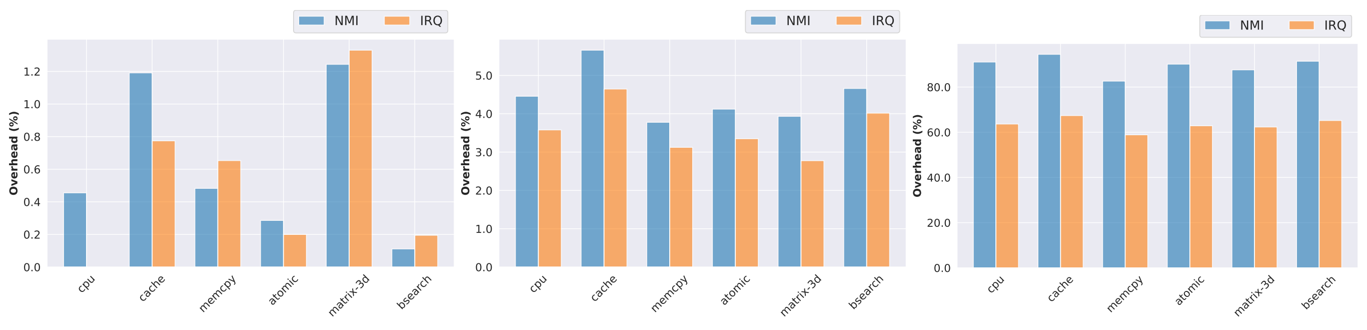
Conversely, some configuration variables are exposed as module parameters to simplify the configuration. Some are the *sampling period* (defined in the elapsed clock cycles domain) and the *PMI vector line*. Beyond the capabilities discussed in Section 4.4, we have introduced proc files to access and query tool parameters such as the number of collected samples. All the results are averaged over 5 different runs.

5.1 | Efficient Collection of PMU Data

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. In this experiment we do not keep collected data buffered, since our focus is to exclude the cost of managing memory buffering and to only focus on the overhead of the two compared interrupt-based schemes.

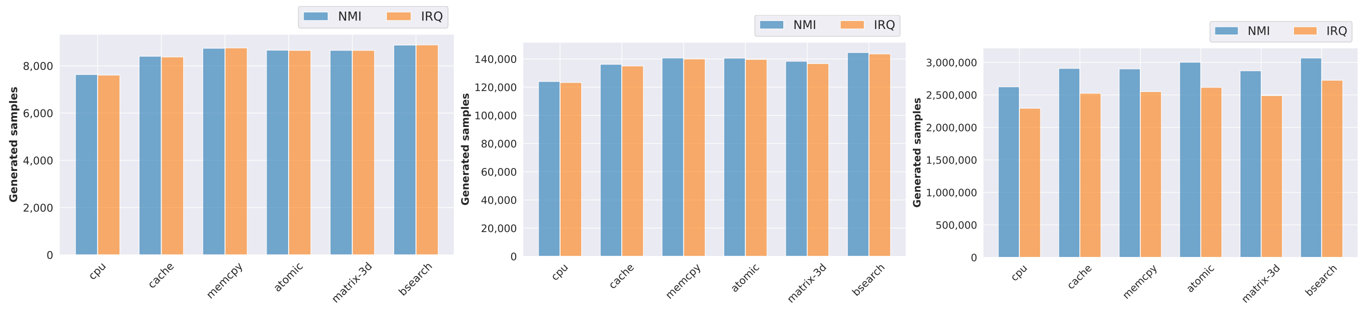
Figure 6 reports the overhead increment of both solutions compared to an execution where no sampling method is enabled. As seen in Figure 6a, a larger sampling period does not make any of the two solutions outperform the other entirely. It is an expected result, as this low-frequency generation of samples 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 7. For lower frequencies, both techniques collected a comparable amount of samples. However, at higher rates (see Figure 7c), 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



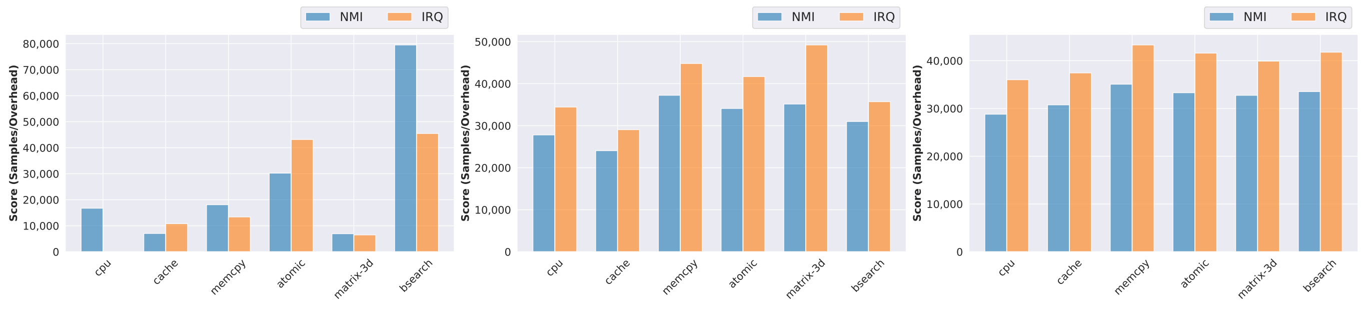
(a) Sampling Period: 2^{20}. **(b)** Sampling Period: 2^{16}. **(c)** Sampling Period: 2^{12}.

FIGURE 6 Workload overhead comparison between IRQ- and NMI-based PMIs.



(a) Sampling Period: 2^{20}. **(b)** Sampling Period: 2^{16}. **(c)** Sampling Period: 2^{12}.

FIGURE 7 Samples generation comparison between IRQ- and NMI-based PMIs.



(a) Sampling Period: 2^{20}. **(b)** Sampling Period: 2^{16}. **(c)** Sampling Period: 2^{12}.

FIGURE 8 Efficiency between IRQ- and NMI-based PMIs.

NMI handling, which cannot prevent PMUs from starting their monitoring before leaving the PMI routine[13]. Therefore, the higher count is mostly related to the PMI routine monitoring itself, which is a clear form of Heisenmonitoring[29].

Figure 8 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.

[13]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 *defreezing* the HPCs.

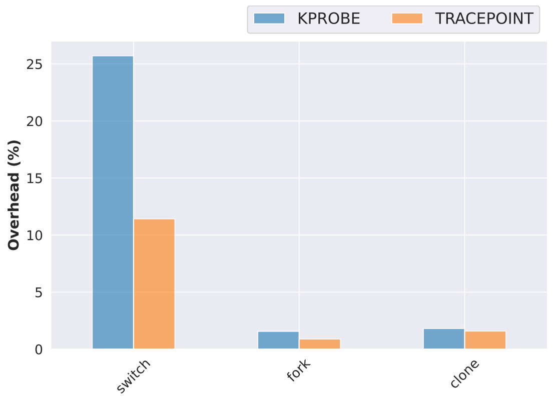


FIGURE 9 Runtime overhead of context-switch intensive workloads with active system hooks.

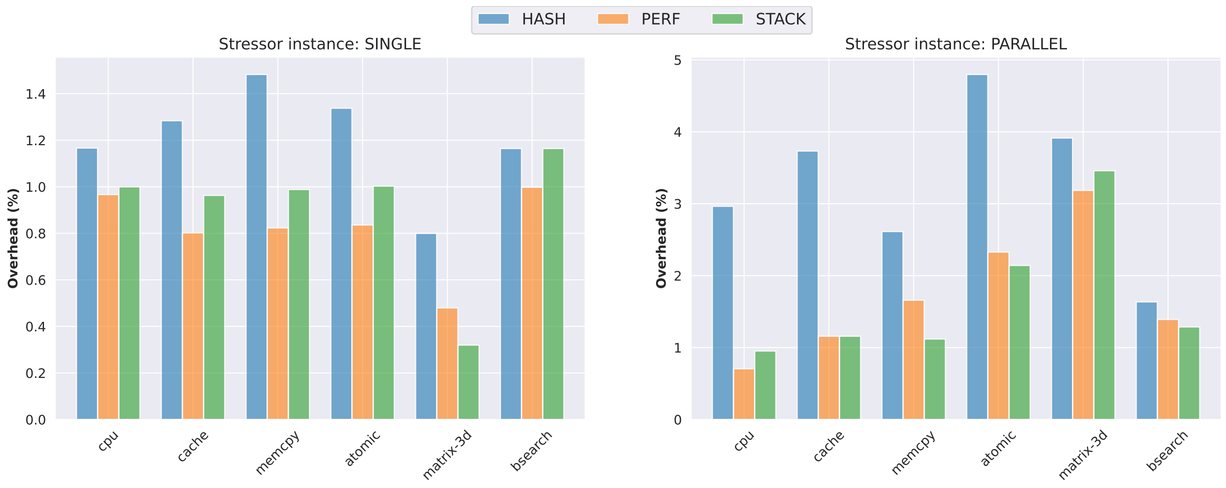


FIGURE 10 Comparison among different approaches to provide per-process private data.

5.2 | Discriminating the Profiling Domain

In this experiment, we evaluate the performance implications of the techniques proposed in Section 4.2 to discriminate the profiling domain. We first focus on the overhead associated with tracepoints and kprobes, excluding data acquisition and buffering. 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 9 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 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 on each available CPU core (*PARALLEL* in the plots). In this experiment, the sampling period has been set to 2^{14}, while the PMI management is based on NMIs and context-switch detection is based on tracepoints. Figure 10 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 overhead.

We then focused on the different buffering strategies. In this experiment, whose results are reported in Figure 11, we assessed the overhead of each discussed method. The results show that the non-blocking queue incurs the highest overhead in most configurations due to the high degree of contention generated by the multiple readers. The *per-CPU* solution introduces the most

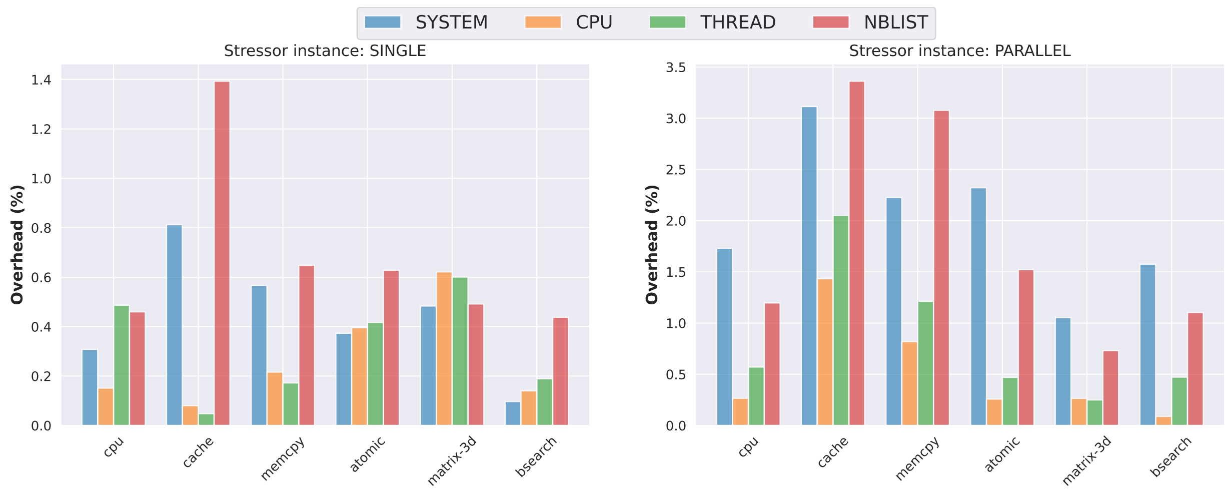


FIGURE 11 Comparison among different buffering policies to collect process profiling data.

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.

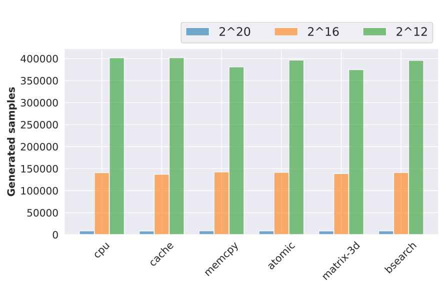
Conversely, switching the buffer reference upon a context switch negatively affects cache performance. Such an effect is more noticeable in the right plot, where the number of participating processes demands an intense scheduling activity. Analogously, the drawbacks of the global buffer instance can be observed when the writers increase and a synchronisation mechanism comes into place, which needs to be used to permit a single writer at any time (we refer this configuration to as SYSTEM in the plot, it represents a system-wide usage of the buffer with multiple possible CPUs attempting write operations). We recall that test sets the sampling period to 2^{12} generating a high number of writing requests. Finally, we assessed the non-blocking circular queue implementation by using the per-CPU policy. As we can observe, this solution 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 clearly affects highly synchronized and cache-unfriendly accesses such as the ones provided by SYSTEM and NBLIST arrangement. Conversely, our mostly non-blocking queue guarantees enhanced progress for PMI management and cache-friendliness, delivering lower overheads in both per-CPU and per-thread configuration.

We have also carried out a comparison with the built-in perf tool, whose results are reported in Figure 12. 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 approaches, 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, and that depending on the objective for which PMU data are used may be unacceptable—like for the case of security based on the hardware footprint left by the running software[4].

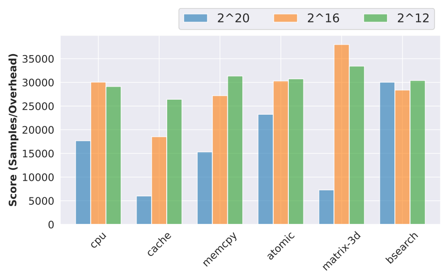
Similarly to the IRQ/NMI test, we evaluated the overhead (Figure 12c) and the number of generated samples (Figure 12a). We can observe that the perf solution introduces a lower cost at the highest frequency rate. However, the number of generated samples is way smaller at the fastest frequency due to the auto-tuning frequency policy. Figure 12b shows the efficiency of the perf solution computed as the amount of generated data over the overhead execution percentage.

5.3 | Accessing Profiling Data from User Space

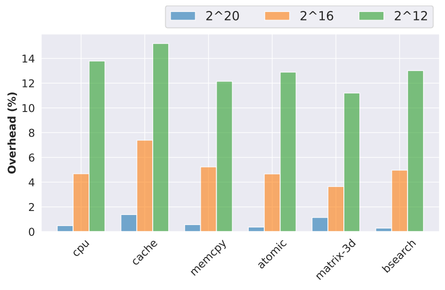
In this last experiment, we evaluate the overhead associated with the different techniques explored in Section 4.4 to retrieve performance data from kernel space. We evaluated their performance by reading a large buffer allocated within a kernel module



(a) Samples generation with active perf profiling 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 12 Perf tool evaluation at different sampling periods and 8 hardware events monitored

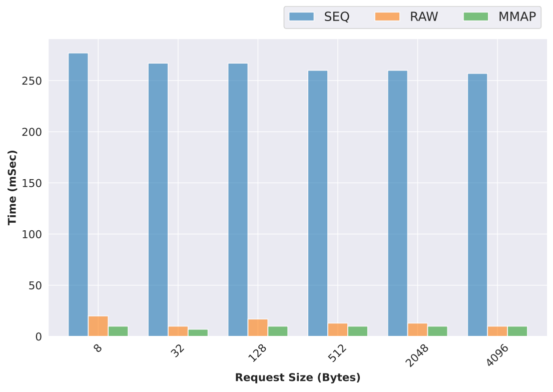


FIGURE 13 Time to fully read kernel level data via mmap- and VFS-based solutions.

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.

As shown in Figure 13, the VFS solution based on the seq_file (SEQ) 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, denoted as RAW, (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 (MMAP) 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 as a part of the synchronisation logic, making the overall implementation more complex and coupled between kernel space and userspace.

5.4 | Data Related to a Case Study

To complete our analysis, we focus on an application area that has recently taken advantage from the exploitation of HPCs. This application is archetypal of domains that may exhibit advanced capabilities in terms, e.g., of functional operations supported via hardware rather than software. As we mentioned before, HPCs can have for these applications positive impact on both performance and power usage.

In particular, we report data related to HPC-based incremental checkpointing in a speculative parallel computing system for discrete event simulation, called ROOT-Sim[62]. In this study, we exploited the per-thread buffering approach, and report data showing minimal intrusiveness of the HPC-based support, where the identification of the parts of a memory object (a simulation object) that have been dirtied by the execution of software does not require any software instrumentation. At the same time, any HPC sample that does not relate to the actual access to the segment where the software object is allocated in memory is discarded when the checkpointing agent exploits data. We recall that these samples might appear just because of aspects related to the kernel-level interference problem we discussed in Section 4.2.2.

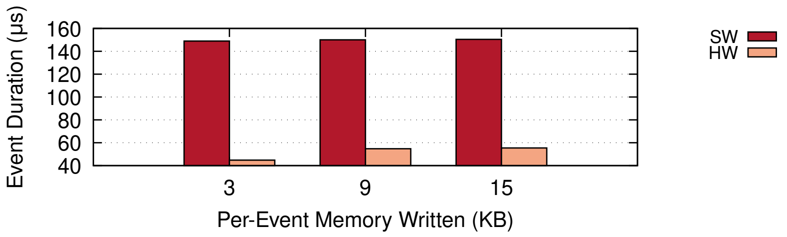


FIGURE 14 Case Study: Event processing duration in an HPC- and software-based incremental checkpointing for speculative simulation.

We have relied on a configuration of the well-known PHold benchmark[63], explicitly embedding parameterizable memory operations patterns. The execution of a simulation event includes read/write mode access to a memory contiguous buffer—implementing the state of the simulation object in the benchmark—of a given size S. In this benchmark, large values of 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 simulation event execution. 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. However, we have adopted an approach where the update operations are based on memory-block writing machine instructions, such as STOS.

We report data comparing the performance of the HPC-based solution for the detection of the updated memory areas, and of an implementation based on software instrumentation of the memory write instructions. As shown in Figure 14, tracing write memory accesses via HPCs through our architecture enables a definitive reduction of the actual latency for processing simulation events.

6 | CONCLUSIONS

In this article, we have presented several techniques that allow leveraging the PMU facilities offered by modern off-the-shelf CPUs to gather hardware-footprint data at runtime. Unlike traditional approaches that exploit such data in a post-mortem analysis, we have discussed the performance impact of the various strategies for moving to online data collection. Our experimental assessment has shown the effects of the strategies and implementations we explored on the application's performance in different workload scenarios. The outcomes show that, if carefully exploited, our techniques could allow for the selection of a practical tradeoff between accuracy, performance, and ease of use, enabling the exploitation of PMUs for, e.g., self-adaptive autonomic optimisation.

ACKNOWLEDGMENTS

This work has been supported by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di "campioni nazionali" di R&S (M4C2-19) - Next Generation EU (NGEU).

References

1. Yasin Ahmad. A Top-Down method for performance analysis and counters architecture. In: *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS. Piscataway, NJ, USA: IEEE 2014 (pp. 35–44).

2. O'Brien Kenneth, Pietri Ilia, Reddy Ravi, Lastovetsky Alexey L., Sakellariou Rizos. A Survey of Power and Energy Predictive Models in HPC Systems and Applications. *ACM Comput. Surv.*. 2017;50(3):37:1–37:38.

3. Bircher William Lloyd, John Lizy K.. Complete System Power Estimation Using Processor Performance Events. *IEEE Transactions on Computers*. 2012;61(4):563-577.

4. Carnà Stefano, Ferracci Serena, Quaglia Francesco, Pellegrini Alessandro. Fight Hardware with Hardware: System-wide Detection and Mitigation of Side-Channel Attacks using Performance Counters. *Digital Threats: Research and Practice*. 2022.

5. Carnà Stefano, Ferracci Serena, De Santis Emanuele, Pellegrini Alessandro, Quaglia Francesco. Hardware-Assisted Incremental Checkpointing in Speculative Parallel Discrete Event Simulation. In: Mustafee Navonil, Bae Ki-Hwan G, Lazarova-Molnar Sanja, Rabe Markus, Szabo Claudia, eds. *Proceedings of the 2019 Winter Simulation Conference*. WSC. Piscataway, NJ, USA: IEEE 2019 (pp. 2759–2770).

6. Akiyama Soramichi, Hirofuchi Takahiro. 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. New York, NY, USA: ACM 2017 (pp. 1–8).

7. Kephart Jeffrey O, Chess David M. The Vision of Autonomic Computing. *Computer*. 2003;36(1):41–50.

8. Hassan Shenin, Al-Jumeily Dhiya, Hussain Abir Jaafar. Autonomic Computing Paradigm to Support System's Development. In: *Proceedings of the 2nd International Conference on Developments in eSystems Engineering*. DESE. Piscataway, NJ, USA: IEEE 2009 (pp. 273–278).

9. Ammons Glenn, Ball Thomas, Larus James R. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. *ACM SIGPLAN Notices*. 1997;32(5):85–96.

10. Basu Kanad, Hussain Suha Sabi, Gupta Ujjwal, Karri Ramesh. COPPTCHA: COPPA Tracking by Checking Hardware-Level Activity. *Transactions on Information Forensics and Security*. 2020;15:3213–3226.

11. Ozcelik Burcu, Yilmaz Cemal. Seer: A Lightweight Online Failure Prediction Approach. *IEEE Transactions on Software Engineering*. 2016;42:26–46.

12. Suciu Alin, Banescu Sebastian, Marton Kinga. Unpredictable Random Number Generator Based on Hardware Performance Counters. In: *Digital Information Processing and Communications*. Springer Berlin Heidelberg 2011 (pp. 123–137).

13. Das Sanjeev, James Kedrian, Werner Jan, Antonakakis Manos, Polychronakis Michalis, Monrose Fabian. A Flexible Framework for Expediting Bug Finding by Leveraging Past (Mis-)Behavior to Discover New Bugs. In: *Annual Computer Security Applications Conference*. ACSAC '20. New York, NY, USA: Association for Computing Machinery 2020 (pp. 345–359).

14. Weaver Vincent M, Terpstra Dan, Moore Shirley. Non-determinism and overcount on modern hardware performance counter implementations. In: *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS. Piscataway, NJ, USA: IEEE 2013 (pp. 215–224).

15. Intel Corporation . Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation2021.

16. AMD64 Technology . AMD64 Architecture Programmer's Manual: Volumes 1-5. Advanced Micro Devices, Inc2021.

17. Dean J, Hicks J E, Waldspurger C A, Weihl W E, Chrysos G. ProfileMe: Hardware Support for Instruction-level Profiling on Out-of-order Processors. In: *Proceedings of 30th Annual International Symposium on Microarchitecture*. MICRO. Piscataway, NJ, USA: IEEE 1997 (pp. 292–302).

18. Bitzes Georgios, Nowak Andrzej. *The overhead of profiling using PMU hardware counters*. Report 2014: CERN openlab; 2014.

19. Mytkowicz Todd, Diwan Amer, Hauswirth Matthias, Sweeney Peter F. Understanding Measurement Perturbation in Trace-based Data. In: *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*. IPDPS. Piscataway, NJ, USA: IEEE 2007 (pp. 1–6).

20. Levon John, Elie Philippe. *Oprofile: A system profiler for Linux.* 2020.

21. Pettersson Mikael. *Perfctr: Linux performance monitoring counters kernel extension.* 2010.

22. Eranian Stéphane. Perfmon2: a flexible performance monitoring interface for Linux. In: *Proceedings of the 2006 Ottawa Linux Symposium*. Ottawa Linux Symposium 2006.

23. Mucci Philip J, Browne Shirley, Deane Christine, Ho George. *PAPI: A portable interface to hardware performance counters.* : University of Tennessee; 1999.

24. Drongowski Paul J. *An introduction to analysis and optimization with AMD CodeAnalyst™ Performance Analyze.* : Advanced Micro Devices, Inc.; 2008.

25. Intel Corporation . Intel® VTune™ Profiler User Guide. Intel Corporation2009.

26. Adhianto L, Banerjee S, Fagan M, et al. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and computation: practice & experience.* 2009;22(6):685–701.

27. Treibig Jan, Hager Georg, Wellein Gerhard. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In: *Proceedings of the 39th International Conference on Parallel Processing Workshops*. ICPPW. Piscataway, NJ, USA: IEEE 2010 (pp. 207–216).

28. Weaver Vincent M. Linux perf_event features and overhead Second International Workshop on Performance Analysis of Workload Optimized Systems2013.

29. Neville-Neil George V. The Observer Effect. *Communications of the ACM.* 2017;60(8):29–30.

30. Merkel Andreas, Stoess Jan, Bellosa Frank. Resource-conscious scheduling for energy efficiency on multicore processors. In: *Proceedings of the 5th European conference on Computer systems*. EuroSys. New York, NY, USA: Association for Computing Machinery 2010 (pp. 153–166).

31. Singh Karan, Bhadauria Major, McKee Sally A. Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architecture News.* 2009;37(2):46–55.

32. Wang Zheng, O'Boyle Michael F P. Mapping Parallelism to Multi-cores: a Machine Learning Based Approach. In: *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. PPoPP. New York, NY, USA: Association for Computing Machinery 2009 (pp. 75–84).

33. Shahid Arsalan, Fahad Muhammad, Manumachu Ravi Reddy, Lastovetsky Alexey L.. Energy Predictive Models of Computing: Theory, Practical Implications and Experimental Analysis on Multicore Processors. *IEEE Access.* 2021;9:63149–63172.

34. Shahid Arsalan, Fahad Muhammad, Reddy Ravi, Lastovetsky Alexey. Additivity: A Selection Criterion for Performance Events for Reliable Energy Predictive Modeling. *Supercomputing Frontiers and Innovations.* 2017;4(4):50–65.

35. Shahid Arsalan, Fahad Muhammad, Manumachu Ravi Reddy, Lastovetsky Alexey L.. Improving the accuracy of energy predictive models for multicore CPUs by combining utilization and performance events model variables. *J. Parallel Distributed Comput..* 2021;151:38–51.

36. Zellweger Gerd, Lin Denny, Roscoe Timothy. So many performance events, so little time. In: *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '16. New York, NY, USA: ACM 2016.

37. Lv Yirong, Sun Bin, Luo Qingyi, Wang Jing, Yu Zhibin, Qian Xuehai. CounterMiner: Mining big performance data from hardware counters. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. MICRO '18. Piscataway, NJ, USA: IEEE 2018.

38. Neill Richard, Drebes Andi, Pop Antoniu. Fuse. *ACM transactions on architecture and code optimization.* 2017;14(4):1–26.

39. Banerjee Subho S, Jha Saurabh, Kalbarczyk Zbigniew, Iyer Ravishankar K. BayesPerf: minimizing performance monitoring errors using Bayesian statistics. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM 2021 (pp. 832–844).

40. Bhatia Sapan, Kumar Abhishek, Fiuczynski Marc E, Peterson Larry. Lightweight, High-resolution Monitoring for Troubleshooting Production Systems. In: *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*. OSDI. Cambridge, MA, USA: Usenix Association 2008.

41. Bare Keith A, Kavulya Soila, Narasimhan Priya. Hardware Performance Counter-based Problem Diagnosis for e-commerces Systems. In: *Proceedings of the 2010 IEEE Network Operations and Management Symposium*. NOMS. Piscataway, NJ, USA: IEEE 2010 (pp. 551–558).

42. Lachaize Renaud, Lepers Baptiste, Quéma Vivien. MemProf: A Memory Profiler for NUMA Multicore Systems. In: *Proceedings of the 2012 USENIX Annual Technical Conference*. USENIX ATC. Cambridge, MA, USA: Usenix Association 2012 (pp. 53–64).

43. Marathe Jaydeep, Mueller Frank. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In: *Proceedings of the 11th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. PPoPP. New York, NY, USA: Association for Computing Machinery 2006 (pp. 90–99).

44. Dashti Mohammad, Fedorova Alexandra, Funston Justin, et al. Traffic Management: a Holistic Approach to Memory Placement on NUMA Systems. In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. New York, NY, USA: ACM Press 2013 (pp. 381–394).

45. Molka Daniel, Schöne Robert, Hackenberg Daniel, Nagel Wolfgang E. Detecting Memory-Boundedness with Hardware Performance Counters. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE. New York, NY, USA: ACM 2017 (pp. 27–38).

46. Das Sanjeev, Werner Jan, Antonakakis Manos, Polychronakis Michalis, Monrose Fabian. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In: *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. SP. Piscataway, NJ, USA: IEEE 2019 (pp. 20–38).

47. Herath Nishad, Fogh Anders. These are Not your Grand Daddys CPU Performance Counters: CPU Hardware Performance Counters for Security Black Hat USA 20152015.

48. Zhou Hongwei, Wu Xin, Shi Wenchang, Yuan Jinhui, Liang Bin. HDROP: Detecting ROP Attacks Using Performance Monitoring Counters. In: Huang Xinyi, Zhou Jianying, eds. *Information Security Practice and Experience*. Lecture Notes in Computer Science, vol. 8434: Cham, Germany: Springer International Publishing 2014 (pp. 172–186).

49. Nomani Junaid, Szefer Jakub. 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*. HASP. New York, NY, USA: ACM 2015 (pp. 1–4).

50. Gruss Daniel, Maurice Clémentine, Wagner Klaus, Mangard Stefan. Flush+Flush: A Fast and Stealthy Cache Attack. In: Caballero Juan, Zurutuza Urko, Rodríguez Ricardo J, eds. *Detection of Intrusions and Malware, and Vulnerability Assessment*. Lecture Notes in Computer Science, vol. 9721: Cham, Germany: Springer International Publishing 2016 (pp. 279–299).

51. Canella Claudio, Van Bulck Jo, Schwarz Michael, et al. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security. Cambridge, MA, USA: USENIX Association 2019 (pp. 249–266).

52. Pierce Cody. Detecting Spectre And Meltdown Using Hardware Performance Counters https://www.elastic.co/blog/detecting-spectre-and-meltdown-using-hardware-performance-countersAccessed: 2022-4-15; 2018.

53. Li Congmiao, Gaudiot Jean-Luc. Detecting Spectre Attacks Using Hardware Performance Counters. *IEEE Transactions on Computers*. 2021:1–1.

54. Lipp Moritz, Schwarz Michael, Gruss Daniel, et al. Meltdown: Reading Kernel Memory from User Space. In: *Proceedings of the 27th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association 2018 (pp. 973–990).

55. Gruss Daniel, Lipp Moritz, Schwarz Michael, Fellner Richard, Maurice Clémentine, Mangard Stefan. KASLR is Dead: Long Live KASLR. In: *Engineering Secure Software and Systems*. Springer International Publishing 2017 (pp. 161–176).

56. Intel Corporation . 8th and 9th Generation Intel Core Processor Family Intel® Core™ Processor Families and Intel® Xeon® E Processor Families. Intel Corporation2020.

57. Herlihy Maurice, Shavit Nir. On the Nature of Progress. In: Fernàndez Anta Antonio, Lipari Giuseppe, Roy Matthieu, eds. *Principles of Distributed Systems*. Lecture Notes in Computer Science, vol. 7109: Berlin Heidelberg, Germany: Springer International Publishing 2011 (pp. 313–328).

58. Diener Matthias, Madruga Felipe, Rodrigues Eduardo, et al. 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*. HPCC. Piscataway, NJ, USA: IEEE 2010 (pp. 491–496).

59. Calandrino John M, Anderson James 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*. ECRTS. Piscataway, NJ, USA: IEEE 2009 (pp. 194–204).

60. Di Gennaro Ilaria, Pellegrini Alessandro, Quaglia Francesco. 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)*. IEEE 2016 (pp. 291–300).

61. King Colin Ian. *Stress-ng: A stress-testing Swiss army knife.* 2019.

62. Pellegrini Alessandro, Vitali Roberto, Quaglia Francesco. The ROme OpTimistic Simulator: Core Internals and Programming Model. In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. SIMUTOOLS. Brussels, Belgium: ICST 2012 (pp. 96–98).

63. Fujimoto Richard M. Performance of Time Warp Under Synthetic Workloads. In: Nicol David, ed. *Distributed Simulation*. PADS '90. San Diego, CA, USA: Society for Computer Simulation International 1990 (pp. 23–28).

How to cite this article: Stefano Carnà, Romolo Marotta, Francesco Quaglia, and Alessandro Pellegrini (2023), Strategies and Software Support for the Management of Hardware Performance Counters, *Software: Practice and Experience*, *2023;00:1–29*. https://doi.org/10.1002/spe.3236