SAPIENZA
UNIVERSITÀ DI ROMA

Master Thesis in

ENGINEERING IN COMPUTER SCIENCE

HOP - Hardware-based Online Profiling of multi-threaded applications via AMD Instruction-Based Sampling

Advisor
Prof. Bruno Ciciani

Candidate
Stefano Carnà

Co-Advisor
Dr. Simone Economo

External Advisor
Prof. Francesco Quaglia

Academic Year 2016/2017

Acknowledgment

I would like to express my sincere and deepest appreciation to all the people who contribute for this thesis to come true.

I would like to express my gratitude to my thesis advisor Prof. Bruno Ciciani for the patient guidance, inspiration and advice he provided throughout my time as his student.

I am deeply grateful to Prof. Francesco Quaglia who encouraged me to pursue this project and taught me that there is always something to find out. His marvellous courses let me finally undertake the path I feel right to me.

I owe my deepest gratitude to Simone, my mentor and second reader of this thesis. I am gratefully indebted to him for his valuable comments on this work that steered me in the right the direction whenever I needed it. I hope we will able to work again together in the future.

I would like to acknowledge the priceless input of Joseph Lee Greathouse, who gave a considerable contribution to this work in the designing phase. I will never forget his precious tricks and clues in the long email he sent to me.

The last but not the least, I would also like to show my gratitude to the HPDCS group whose guys were always ready to help me whenever I run into trouble. You made me enjoy these months despite the the arduous period.

This page has been intentionally written in Italian because the reader appreciation is more important than the message itself.

Chi mi conosce bene sa che sono una persona molto introversa e soprattutto restia ad esternare i propri sentimenti. Ma poiché laurearsi implica anche maturare, oggi mi tocca, ed è così che per questa occasione, con un po' di orgoglio e timidezza, proverò ad esprimere il mio affetto, cercando di essere un po' meno "ingegnere".

Vorrei iniziare ringraziando la mia famiglia, senza la quale non avrei mai neppur cominciato questa carriera. Consegno simbolicamente a mio padre e mia madre questa tesi e questa laurea, frutti di un lungo e prezioso cammino universitario che entrambi mi hanno permesso di percorrere e concludere, sostenendomi sia economicamente che emotivamente.
Grazie a mio padre, che mi ha sempre consigliato nelle scelte più importanti della vita, incoraggiandomi ad affrontare ogni difficoltà con la sua stessa forza e volontà d'animo.
Grazie a mia madre: amica, cuoca, infermiera e consigliera; la persona che ha fatto di me ciò che sono, il mio punto di riferimento da sempre e per sempre.
Grazie ai miei fratelli, Chiara e Alessandro, sempre pronti ad ascoltarmi e a condividere le mie vittorie e le mie sconfitte. A Chiara, la mia guerriera, perché ha reso la nostra convivenza decisamente più allegra ed accogliente e perché ogni volta che ho avuto bisogno, lei c'era! Al "piccolo" — che ormai tanto piccolo non è — Alessandro, che si è appassionato da subito ai miei progetti e che, nonostante i chilometri che ci separano, sento sempre presente.

Grazie a nonna Viola e a zio Cocò, che anche se oggi non possono essere qui a festeggiare con me, spero che mi stiano guardando da Lassù sorridendo e che siano orgogliosi di me e del traguardo raggiunto.
Grazie ai miei zii per l'affetto che mi hanno dimostrato, per essere sempre stati fieri e orgogliosi di me e per avermi fatto sentire il loro "Ingegnere" anche quando questa avventura era appena cominciata.
Grazie a tutti i miei cugini, a Cau e a Vicenzo, che hanno sempre riposto fiducia in me, mettendo alla prova le mie capacità e affidandosi a tutti i miei pareri e consigli.

Grazie ai miei compagni di sempre, per le chiacchierate, le risate e i momenti spensierati trascorsi insieme.
Grazie ai miei cari "colleghi": per avermi aiutato ad affrontare il percorso universitario in modo serio ma non serioso, con costanza ma senza eccessivi affanni. Perchè siete stati semplicemente gli amici di cui ho avuto bisogno.

Un grazie particolare alla mia ragazza Idabelle per il sostegno e l'amore donatomi in questi anni. Perchè mi sei sempre stata vicina consigliandomi nelle scelte importanti, condividendo i momenti di gioia come quelli più difficili. Per essere stata la persona che fin da subito ha reso l'inizio di questo percorso più allegro e spensierato. Perchè mi hai dato modo di sperimentare le mie doti culinarie e di sorridere sempre anche quando non riuscivo al meglio nell'intento. Per la tua gentilezza e la tua ospitalità nei freddi weekend invernali e nei caldissimi giorni d'estate. Perchè ti sei sempre presa cura di me quando ne ho avuto bisogno. Grazie per esserci.

Stefano Carnà

To Uncle Cocò and Grandma Viola,
for the silent backing you gave to me.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

The number of transistors in a dense integrated circuit doubles approximately every two years.

— Gordon Moore, Co-founder of Intel (1965)

Simple computing architectures are just memories. The Moore's law is the empirical rule that predicts the electronic development for over 40 years, so accurately, that lays the "roadmap" foundations for most of the semiconductor manufacturers. Until a few decades ago, the computer architecture evolution was mainly based on the operating frequency growth, namely the speed of the processor. However, in 2003 the gap between the performance achieved by processors and the Moore's law came up, diverting the computer progression to other ways. Moore's law is still sound. The more and more availability of transistors has been exploited to implement more sophisticated architectures, able to take advantage of polished capabilities rather than simple *brute force*.

Nowadays, processors are based on *supercalar* architectures as well as out-of-order execution engines. These not only allows achieving better instructions per clock (IPC) than scalar solutions, but also optimize the code execution through the employment of further mechanisms such as speculative computation.

Besides these *core improvements*, companies walks the road toward the process parallelism. This led to multi-core processors, which include more computing cores on the same chip. However, the increasing power capability on processors and the memory speed did not go hand in hand. Accessing the memory still represents a bottleneck during computation because CPU-core processing is far faster than memory operations. To overcome this speed limitation, memory elements have been directly implemented on chip such that the communication with the processing units is subjected to smaller latency. These memories are know as *cache memories* which are so pervasive

that their structure have been further enhanced by providing several layers, resulting, along with the main memory, in a sophisticated hierarchy.

The high number of cores sharing memory in a single system bumped in another memory issue. As a matter of fact, the memory cannot easily handle the concurrent requests by all the cores thus becoming the main performance bottleneck in different scenarios. Non Uniform Memory Access (NUMA) systems were born from the need of coping with this ineptitude. Such systems are formed by a set of nodes which cooperate, sharing computational power and memory resources to carry out advanced system management and problem resolution.

Heterogeneous computing refers to systems that take advantage of dedicated cores to carry out specific tasks. An example of such solution is the combined work of CPUs and GPUs. They are suitable for different kinds of calculations and mixing their capabilities allows achieving several goals such as efficiency, higher performance and less power consumption.

In such a complex world, modern software, for its part, tries to benefit— in the best possible way— from the underlying hardware facilities without exposing too many hardware-level details to developers.
Yet, how is it possible to find out the reason of a program behaviour when it does not act as we expected? Simple: use a profiler!

Profilers are tools specially designed for observing the execution of an application or the entire system with the aim of a profile creation. Such a profile holds the information gathered during the investigation and can be fed to external tools for further analysis.
Most of the profilers are based on software techniques which, though capable of revealing a lot of execution information, may just observe high-level events also incurring in a significant overhead.

Most of the modern processors include within their architecture some specialized elements used to gather information about what is going on, at the hardware level, during code execution. Such elements are known as Performance Monitor Units and allow to understand the reasons of several issues that may not directly recognized at higher level. Enhancing profiling tools with this support may lead to a low-overhead and more transparent software analysis.

The remainder of this thesis is structured as follows. Chapter 2 provides an overview of the profiling techniques focusing on hardware instrumentation reached by means of the on-chip performance monitor units. It analy-

ses existing implementations—along with their advantages and drawbacks—and extends its inspection to the most relevant works in the literature which rely on such a hardware support. Chapter 3 is the core of this thesis and presents our hardware-based profiler illustrating its capabilities along with its design qualities. Additionally, it introduces several issues to tackle when directly working with Instruction Based Sampling (IBS). Chapter 4 provides the reader with an experimental justification of the goodness of our solution in terms of overhead, accuracy, and efficiency, also commenting some IBS intrinsic characteristics we observed during our experimentation. Finally, Chapter 5 concludes and sums up this thesis. It discloses how our proposal can be enhanced for an extended support and possible directions for future work.

CHAPTER 2

Program Profiling

The more you know, the more you realize you know nothing.
— Socrates

Program profiling is the fine art of observing the behaviour of an application during its execution. The analysis performed on such an examination may identify several problems such as CPU stalls, memory leaks, blocking I/O operations, inefficient memory usage and performance bottlenecks. The employment of profiling tools not only allows to point out performance issues, but it also provides a primary form of debugging and security audit. Although there are alternative ways to perform application analysis, program profilers can carry out an optimal examination of code and data in a program in terms of accuracy and overhead. Indeed, in order to study the program of interest, some extra work has to be carried out, which necessarily leads to additional resource usage. Contrary to some rudimentary techniques that insert *print* operations to identify the execution flow or the occurrence of some event, the profiler can insert hotspots in specific points of the code in a less intrusive fashion. One of the reasons that nowadays make a profiler an essential tool for performance analysis is the increasing complexity of computer architectures as well as the modern trend to simplify programming languages. High-level languages and programming frameworks represent an essential part of software development, and though they relieve developers of low-level aspects such as memory management (pointers, allocation and deallocation), the code may not be optimized for hardware facilities like *cache memory*. Generally speaking, the profilers deal with just the monitoring of the execution flow and the data collection of an application and are further supported by a set of tools which perform the actual analysis. Relying on such an analysis, a user may operate on possible *not-well-written* code sections, enhancing the overall application performance.

2.1 Profiling Techniques

Instrumentation, as we intend it in this work, is the technique which the majority of profilers rely on. A profiling tool may adopt either software or hardware instrumentation according to the scope of its analysis. However, the overhead a profiler produces highly depends on the technique being adopted. While the hardware approach exploits on-chip elements which are code-agnostic, software instrumentation acts on the code being executed by injecting extra instructions. Even though such techniques work at different levels and can answer different kinds of questions, a mixed solution can be adopted in order to produce a more accurate outcome.

2.1.1 Software Instrumentation

Software instrumentation represents the most common used approach. The main way to instrument an application adopting this technique is patching the application itself. Even though this may be more or less transparent to the application developer, it may not be so from the program point of view, just depending on the kind of instrumentation adopted. Software instrumentation can be applied on software at different levels of abstraction: the end-user application, the middleware libraries, or even the operating system itself. This is extremely useful because, depending on the context, it is possible to analyse either the environment in which user applications are running or limiting the instrumentation to the program itself. The instrumentation provides a code patching technique which can be performed at different representation levels of software:

- **Machine code level**: at this level, everything is seen just as pure byte sequences. Both instructions and data lose their semantic nature given that high-level elements (control flow execution and data types) of the related program can be barely identified. Despite this, it results useful for particular tasks like memory analysis since the access pattern is explicit at this level.

- **Byte code level**: some compilers, like just-in-time ones, provide this transitional code form which is in between machine level and source level code. Compared to the previous, at this level it is possible to have some semantic clues about the program control flow and data structures. Moreover, it goes beyond the particular limitations of the underlying architecture by working on hardware-independent code.

- **Source code level**: the program description degree at this level comprises all its internal composition, ranging from the sophisticated

control-flow paths to the advanced data structure metadata. Even though this represents the most precise level concerning program semantics and guarantees the best portability of the instrumentation logic, it requires the availability of the program source code as the target of the instrumentation task.

There are two kind of software instrumentation techniques which depend on when the instrumentation is applied.

Static Instrumentation

Static instrumentation takes place when patching is performed at compile-time [LTCS10] [Pel13]. Though it is not transparent to the application because it acts by directly modifying the final executable image, the cost of the instrumentation is paid only once for all the runs. *Relocation* is one of the main techniques based on static instrumentation. The code is duplicated in different text segments and enhanced with instrumentation logic. A flow branch is put in the original executable to execute the instrumented code in place of the original one, where the latter is located (*instrumentation point*). When the execution reaches an instrumentation point, the flow is redirected to the corresponding instrumented section which, at its end, will deflect the flow to the original code. The number of applied changes to the original code are minimal while all the memory addresses are unmodified. Of course, the presence of two branch instructions for each instrumentation point represents an overhead. Additionally, a requisite is placed on the size of the code being instrumented because it must have instructions large enough to make it possible to replace them with branch instructions, without having to shift subsequent code. Another technique is known as *inlining*. It acts in a completely different way. In fact, instead of duplicating sections, it inserts the instrumentation code directly into the original executable. This allows saving the cost of both diverting the execution flow and the memory required to produce the final instrumented code. As a drawback, it requires to shift code and update all the addresses (where required) due to the new instructions insertion which, may result difficult for indirect memory references that typically don't have sufficient meta-data.

Dynamic Instrumentation

Dynamic instrumentation performs the patching operation at application execution-time (it is also called just-in-time instrumentation). Compared to the static methodology, acting at running time makes the dynamic instrumentation highly impact the performance of the application being instrumented. Most of the available toolkits base their activity on two different

approaches know as *native execution* and *emulated execution*. The native execution is performed by means of an instrumentation virtual machine (VM). This copies the instrumented application in its address space and logically sees the related code as a set of code blocks, which are executed one by one. Before being processed, each block is instrumented and patched such that the last instruction returns control to the VM. Additionally, some precautions are adopted in order to speed up the entire process activity. In particular, the already patched blocks are kept in a cache such that already instrumented blocks can be just retrieved from it. Moreover, the *linking* technique allows sparing the VM dispatcher calls after a block execution by early understanding if the following one is already in cache. In this case, the second block is directly linked to the previous one, and the execution is not interrupted in the middle. The main drawback of this technique is given by the fact that it has to capture all the execution-flow directions such as those provided by branches or even privileged actions (e.g., exception generation). The emulated execution takes advantage of an *intermediate representation* of the code to generalize some information tied to the architecture, obtaining an abstract low-level language (byte-code style). Furthermore, the processor is subjected to a full emulation of its state and elements such as its registers are logically represented. The original application code is never executed, but the VM presents only a compatible version of that such that the control is never released and is therefore more useful in scenarios where applications are never executed natively for security and portability purposes.

2.1.2 Hardware Instrumentation

Hardware instrumentation extends its scope to several domains. A first approach is represented by snooping the electrical signals of the analysed hardware via sophisticated probing instruments. This technique requires either the hardware interface for the attachment of such debugging instruments— which are usually available in embedded system platforms— or a deep knowledge of the instrumented platform so that the user can manually hack it and reconstruct the debugging scenario. However, the hardware instrumentation, as we intended it in this work, identifies all those activities concerning the use of some specialized elements directly provided by processor architectures. Unlike the software approach, hardware instrumentation allows observing the behaviour of the system at low overhead while detecting architectural-level transitions unlikely cough by other method. The following section largely describes this hardware support.

2.2 Performance Monitor Units

Most of the modern processors include within their architecture some specialized counters used to gather information about what is going on, at the hardware level, during code execution. Performance monitor counters (PMCs) are available since very old architecture like Intel Pentium and AMD Athlon and have been largely extended for years. These counters monitor a set of available architecture-defined hardware events and are divided into

- *fixed-event counters* which can be used to observe already defined events type
- *programmable counters* which allow the user to define the event type of interest to be monitored.

PMCs can be configured to work in one of the following modes [Moo02]:

1. *counting*: the counter is incremented each time the related event occurs. As consequence, inspecting the counter after some time potentially provides the exact number of the event occurrences since the counter was enabled. Its value at any time is the result of an aggregation of data because it keep increasing regardless of the code being executed. However, it does not bring extra information related to the event thus the gathered data insufficient in some situation.
2. *sampling*: to enhance the counting mode, a threshold can be programmed such that an interruption can be sent to the execution flow each time the counter overflows. Thereby it is possible to investigate the context that generated that hardware-event.

The common 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` [Dev17][Cor17] instructions or by directly working on the counter using `RDPMC` [Cor17].

Generally speaking, there are two classes of events that can be monitored [Cor17]. Non-architectural events are not standard defined because of the different ways processors are assembled into larger chips, while the architectural ones provide a set of events that are consistent among architectures. Consequently, the second category is more interesting and provides a higher degree of portability. Among the rich set of events that can be monitored, it is possible to observe

1. *time events*: architectures provide a special counter, called Time-Stamp Counter (TSC), which is in charge of incrementing its value for each clock cycle[1];

2. *instructions progress*: a dedicated counter tracks the retired instructions during CPU activity, providing a primary form of processor throughtput[2];

3. *memory data*: caches can be monitored at several levels and counters can be incremented for each miss/hit event;

4. *branches*: a counter can provide further information on branch instructions such that it is possible to notice events such as branch mispredictions or retired branch instructions.

Given this plentiful support for the hardware-events, the accuracy of PMCs has been largely studied and discussed. In [KTC01] Korn et al. conduct an extended study on the primary implementation of the performance counter. To access the PMCs, the authors exploited facilities given by software as *perfex* and *libperfex*. The performance counters accuracy analysis was performed by comparing the data obtained by PMCs and a simulated run over several micro-benchmarks executions. Accuracy highly depends on the kind of used interface as well as the application and the event being measured. A more recent study [ZJH09] conducted by Zaparanuks et al. presents a meticulous study done on performance counter accuracy, also comparing with other similar works. They based the PMU access on PAPI and perfmon softwares (see section 2.3) and matched observed counter events with simulated data provided by a statistical study on the benchmark structure. Authors conclude by highlighting the high inaccuracies reached by some tests performed on different architectures.

Using the PMCs in sampling mode is the context in which this inaccuracy problem is observed with greater probability. In this case, a threshold must be specified so that, when the counter reaches it, the processor can be notified. The way the event is notified on modern systems is by sending an interrupt. To get the instruction associated with the micro-operation that generated that event, one would look at the instruction pointer (IP) register at the time the overflow occurred. However, upon the interruption generation, the IP is saved into the stack and may not represent the actual

[1] Actually, its activity depends on the processor state and, in some cases, it may skip the counting. RDTSC is the x86 instruction to access its contents.

[2] For instructions composed of multiple micro-operations, the counter is incremented when the last micro-op is retired. Note that a REP prefix does not effect the number of times the counter is advanced.

instruction of interest. This problem is largely enhanced in the modern superscalar architectures which take advantage of the use of an out-of-order engine that can execute several operations concurrently[3]. Furthermore, due to the different order of micro-operation execution and instruction retirement, the sampling notification may be late with respect to its generation point. This delay is called *skid*. Stated differently, the triggered event may not be precisely associated with the instruction that generated it, but to one of its neighbours. Indeed, when measuring program efficiency, for instance, cache misses, this produces low-precision results. The skid is one of the problems that led vendor to improve the Performance Monitor Units by extending the PMCs with extra capabilities, also intensifying the hardware complexity. From now on, with the term PMU we refers to entire hardware support including both traditional PMCs and extra elements.

2.2.1 Intel Precise Event-Based Sampling

To go beyond the traditional performance counters precision, Intel extended the PMC support with the Precise Event-Based Sampling (PEBS) [Cor17]. This new support introduces the *precise event* concept over the non-precise events observable by PMCs. Even though PEBS cannot be applied for all the available events but only for a limited subset of them, it provides some new mechanisms that automatically save the hardware context when the counter overflows consequently avoiding a code interruption to gather extra processor information related to the event itself. Note that PEBS does not block the usage of the standard counters, so they can work in a combined manner. Generally speaking, the user should program a counter to increment upon a determinate event generation and contextually should define if it should work with PEBS or not. A first essential element introduced is the *PEBS assist*. It comprises a predefined micro-code that is in charge of saving the hardware context when the counter overflows, bypassing the generation of an interrupt like legacy counter mechanisms. Additionally, it is possible to program the counter in sampling-mode such that when the threshold is met, an interrupt is fired[4]. When gathering the event-related data [5], 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. Each time a sample is produced it is added to the tail which is known as

[3]AMD Fam 10Th processor can have up to 72 operation in-flight at any time

[4]In this case, the interrupt will arrive after the PEBS assist completion. If both the counter-interrupt and PEBS-interrupt are programmed, and they are concurrently ready to be generated, the processor will postpone their execution after PEBS-assist. In this case, two different interrupts are generated.

[5]Data comprises the EIP, EFLAGS and processor general purpose registers.

PEBS index. When the PEBS index reaches that threshold value, an interrupt informs the operating system that the buffer is almost full and a read operation should be conducted.

2.2.2 AMD Instruction-Based Sampling

To overcome the skid issue and other minor problems, AMD relied on a different sampling mode. So far, we saw how hardware counters and advanced units conduct the sampling task on hardware-events. As a more sophisticated support to PMU with the Family 10Th processors (Barcellona), AMD introduced in 2007 the Instruction-Based Sampling feature. IBS is built on the technique described in [DHW+97] and, as the name suggests, it observes instructions instead of events. There is no best method between event-sampling and instruction-sampling, but rather the context defines what solution is more suitable. The former focuses on event generation, limiting the number of observable events to the number of available hardware counters. Thus, instruction metadata brought back upon event generation. The latter performs the sampling on executed instructions by the processor, statistically choosing the instruction to profile during its entire flow and gathering produced information. This method extracts data related to events generated by the sampled instruction.

IBS Specifics

The pipeline structure within the processor is decoupled into two main phases: the *fetch stage* witch retrieves the next instruction to be executed and the *execution stage* that represents the step where the instruction is performed. IBS adapts to this scheme and exposes two different interfaces associated with the same working logic but operating at a different level: the *fetch sampling* and the *execution sampling*. Table 4.1 details the information for both the subsystems [Dev11].

Like Intel PEBS, IBS bases its activities on model-specific registers. The most important MSR is represented by the IBS control register, while others are used to keep data information, thus being accessed upon a new sample generation. Figures 2.1 and 2.2 respectively show the Control Register for IBS fetch and IBS Execution. It is possible to note some common elements between the two registers that are used specifically to configure the sampling activity:

- `Ibs*En`: This bit represents the activation bit, thus as long as it is set the support is working.
- `Ibs*Val`: Whenever a sample is generated this bit is set. Conse-

IBS Fetch	IBS Execution
Completed or Aborted fecth	Branch information
Clock cycles spent on the fetch	Time from tagging to retirement
I-Cache hit or miss	Multiple levels D-Cache hit or miss
I-TLBs hit or miss	Multiple levels D-TLBs hit or miss
Linear and physical instruction address	Linear and physical memory addresses for store/load ops
	Latency to complete the load/store
	If it was a local or remote memory access

TABLE 2.1: *A comparison of binary instrumentation tools*

quently, an interrupt is sent to the CPU[6] so that it can handle the new data. In order to not take samples associated with the interrupt routine code, the sampling stops working while this bit is set.

- `Ibs*CurCnt`: it represents the counter which is incremented according to the sampling logic.
- `Ibs*MaxCnt`: when the counter reaches this threshold value, the next instruction is chosen for profiling.

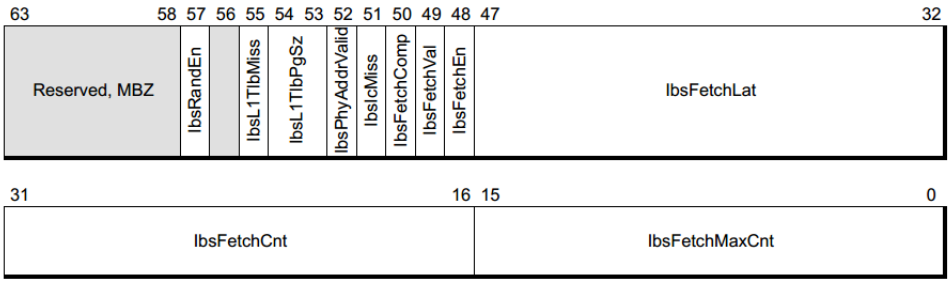

FIGURE 2.1: *Fetch Control Register [Dev17]*

By inspecting the IBS Fetch Control Register, one of the control bits is reserved for randomization[7]. This technique helps to avoid the situation in which the nature of code may lead to a wrong result of the monitored samples. In particular, altough randomization does not provide an all-inclusive solution for certain cases, it allows the sampling activity to exit

[6]AMD does not provide any form of buffering, like Intel does, thus a new sample implies an APIC interrupt.

[7]This bit enables the randomization facility which act on `CurCnt` bits.

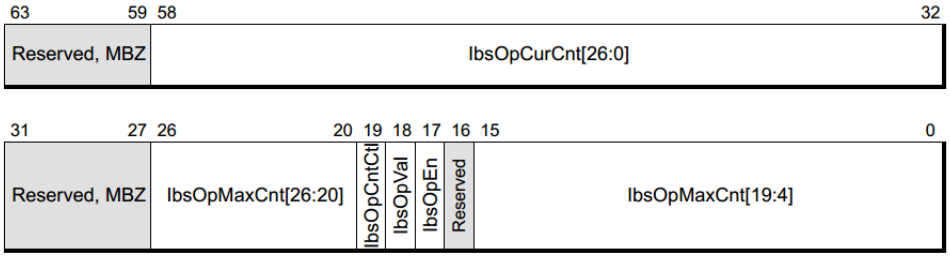

FIGURE 2.2: *Execution Control Register [Dev17]*

from the determinism incurred inside a loop [CVH+10]. This support is not implemented for the IBS Execution part, and randomization is managed at the software level. In Figure 2.2, the scheme shows fewer bits in `MaxCnt` (23) than those used to express the `CurCnt` (27) value. Actually, the `MaxCnt` value is considered left-shifted by 4 so that its logic value spans 27 bits. The reason for this misalignment is to give the possibility of randomizing the last 4 bits of the `CurCnt`. By expressing each time a different value for this nibble [8], it is possible to make the actual value of the sampling period oscillate within 0 and 16 units above its initial value. Last but not least, IBS Execution provides the `IbsOpCntCTL` bit that determines the policy of the sampling activity. In particular, it is possible to base the sampling on counting the dispatched micro-operations or on the elapsed clock cycles. The user must be aware of the fact that if an instruction results in more than one micro-operation, only one of those may be sampled hence potentially leading to unexpected results.

Lightweight Profiling

A very interesting support, called Lightweight Profiling (LWP) [Dev10a], was presented by AMD right after IBS introduction for a more specialized analysis. LWP carries out a more precise activity than hardware counters and IBS, focusing on user-space activity and low overhead profiling. LWP improves speedup by allowing gathering data into a dedicated hardware ring buffer and defining an optional threshold for interruption since it can just work in polling mode. An interesting property is its ability to look at thread context and consequently, it can be used simultaneously by more threads without conflicts. The support is transparent to the end application and collects data uniquely generated by user-space code[9]. Moreover if profiled and not profiled threads exist at the same time, there is no additional overhead to turn off the support for the second ones. The interface between

[8] A nibble is half of a byte.

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

the threads and the LWP is a data structure called the Lightweight Profiling Control Block which can be accessed and configured by the `LLWPCB` and `SLWPCB` instructions. The LWP work can be summarized in five points [Dev10a]:

- *Count*: there are several available event counters that are decreased every time a retired instruction caused a related event[10]. Additionally, it is possible to specify some filters for the observed event driving to a more peculiar monitoring.

- *Gather*: upon counter overflow, LWP collects all the event data. In case of multiple events generation, the behaviour is implementation-dependent. All the events may be sequentially recorded or only one of them can be picked. If there are conflicts due to the presence of multiple events there is a delay in the gathering of the data, some strategies can be put in place to overcome this issue.

- *Store*: in this phase, the event record is saved into the ring buffer located in the process memory address space. The hardware keeps track of the missed events which represent the sample that couldn't be saved because the buffer was full. Moreover, the storing action is not perfectly synchronous with the instruction retirement and may complete later.

- *Report*: this represents the hardware counter step where, if a threshold value is specified, an interrupt is generated[11] and the OS can benefit from this notification to start reading the ring buffer.

- *Reset*: Like IBS this part re-enables a new sampling cycle. Every time a sample is saved, thus leading to increase the buffer HEAD, or a missed event occurs, LWP resets the sampling mechanism.

Unfortunately, LWP didn't have so much success and even its official support by AMD was limited. We didn't found any open-source driver to play with, therefore it is likely that the low interest from users led AMD to remove its support from its processors to take advantage of die space to foster other features.

2.2.3 Overhead

The Performance Monitor Units can be used for activity profiling incurring in a very low resource usage if compared to the software counterpart.

[10] One instruction can cause an unpredictable number of events according to its execution context.

[11] The interruption may be presented much later than the threshold excess.

Nevertheless, the overhead experienced by the examined application substantially depends on the setup of this hardware support. PMUs have been widely studied, starting from old performance hardware counters and reaching the modern IBS and PEBS technologies. Bitzes and Nowak reported in [BN14] a very detailed analysis of PMCs and PEBS performances. Counting mode gives the lowest penalty but, it depends on the number of observed events and running threads during the experiments. Despite an increasing trend when expanding the number of tracked events due to multiplexing, the overhead does not go over 2%. Performance counters are available in a finite number and the set of event concurrently monitored is limited to that quantity. Multiplexing allows spreading the activity over more events. It regulates the counter activity on a programmed events by observing them for a certain amount of time. When the time for the current event expires, the counter value is saved and a new event is programmed to be observed. After some time, its state is loaded into counter so that its activity can restart like no interruption occurred. On Intel Hyper-Threading (HT) compliant architectures the hardware counters are replicated. Thus, by turning off the Simultaneous Multi-Threading support, the HT-related counters can be used by the core, resulting in a double availability which may avoid the multiplexing reducing the experienced overhead. An entirely different scenario is presented when using PMC in sampling mode. Although the primary function of the counter is still counting, the possibility of expressing a threshold value can enhance it such that, upon counter overflow, it is possible to grab context data. Invoking a hardware interrupt introduces a new source of delay. PEBS produces almost the same overhead experienced during tradictional counter sampling activity [BN14]. This behaviour is the result of firing a hardware interrupt upon the sample finalisation. In fact, the software execution is stopped to perform the interrupt handler routine, resulting in more work to be done. The drawback is not limited to the extra code execution, but it implies another backstage phenomenon, i.e.; privilege ring switching, that indeed will cause slowdown because of pipeline flushing and cache pollution [SS10]. Furthermore, we did not found in literature a similar study on AMD architecture, but we got several works which took advantage of the AMD IBS support, consequently limiting the analysis on the developed product [LLQ12] [MV10] [DFF$^+$13]. Although, both PEBS and IBS notify the presence of a new sample via an interrupt-based mechanism, only the former provides a hardware buffer support. Stopping the normal code execution for processing new available data, especially in very high-frequency rates, represents a heavy load for the entire system. PEBS allows an automatized buffering at the hardware level, such that each time a new sample is accessible, instead of sending an interrupt, it is saved into a dedicated memory area. Nevertheless, a threshold value should be set to

make the software aware of the buffer state, then starting reading data from it. PEBS has been relatively overestimated in the literature due it buffering capability. Indeed, it was demonstrated that PEBS assist takes several nanosecond (200-300) to complete, thus causing cache pollution due to fast data writes [AH17]. Moreover, both event and instruction sampling carry on a problem within their nature. Event-sampling, as discussed above, is limited in the number of the possible number of the monitored event at the same time incurring in a multiplexing overhead. Instruction-sampling, following the IBS implementation, lacks instruction selection ability. The instructions are sampled among all in the running code, so that it is not possible to discriminate a particular type, making it more suitable for a general-purpose activity. For instance, if an application wants just to get memory-related instructions, the only way to perform that is by filtering out all unwanted sample types with a clear overhead due to extra sampling collecting.

2.2.4 Portability

One of the big challenge when starting working with PMUs is clearly stated by the not-always-easy interface. There is not a standard way of acting with this supports and each vendor provides its own methods. The complexity of this process is enhanced by the necessity of directly working on hardware registers taking the software to a lower level. Implementing a cross-platform solution may be quite expensive in terms of time because it implies the study of each single architecture reference manual.

Furthermore, PMU is not largely used and even looking for support might not lead to fruitful results. Beside those complications, there is another issue that makes portability an issue. Taking into account a general vendor, for instance Intel, we already discussed that the kind of events that can be observed are twofold: architectural and not-architectural. The former comprises those events that try to keep compatibility among all available models supporting hardware counters and PEBS features. The latter, instead, identifies model specific events which probably are defined for a sub set of model. This implies that basing an application on the availability of determined events must take care of their type if it wants to be able to be executed on several processor models. Moreover, by taking other vendor products into account makes the task as daunting as walking a jungle. Placing side by side the PEBS and IBS support it is actually hard to find a common point to start from. The major problem here is due to the working mode which, in the former, lies in event sampling while in the latter, it is based on instruction-sampling. Even if we can take a step backwards and considering the simpler hardware counters, the available events do not per-

fectly match. To try to overtake the issues mentioned above, some solutions have been proposed.
One of them is *Perfctr*, a low-level interface introduced in 1999 for accessing the hardware performance counters [Pet11]. Although it does not provide advanced features and limits its support to self-monitor analysis and system-wide profiling, many tools adopted perfctr a means for accessing the PMUs. An exciting feature is the ability to use the RDPMC assembly instruction to access the PMC content. Contrary to other instructions like RDMSR, RDPMC can be executed from *not-privileged* level so that the counter can be directly accessed from user-space code. However, the arrival of more sophisticated solutions like *perf_events* supplanted perfctr which nowadays does not represent a valid alternative (its support ceased since 2011 [Pet11]). Even though it provides only little support for PMC, another tool that deserves a brief citation is the Intel Performance Counter Monitor (PCM) library [TWF17]. It is a cross platform tool, but, as the name suggests, it is only compatible with the Intel family processors. Nevertheless, it can configure the performance counters only in counting mode without allowing any further capability such as memory sampling support [SMM16].
PAPI probably represents one of the best project in this field. It is an API for accessing hardware counters within processors [BDG+00] and exposes a high-level interface for a plain activity and a low-level interface to perform more-advanced tasks. This layer in charge of translating actions from low-level to hardware counters depending on the underlying operating system. On Linux, they communicate by means of the perf_event interface. A general PAPI overview shows that its behaviour follows the already discussed procedure:

1. defining the hardware-events of interest,
2. gathering the sample,
3. informing the system via an interrupt.

These steps are tunable thanks to a parametric setup. Moreover, increasing support for PMUs on modern processors aroused the research in this field consequently improving the PAPI capabilities [LMW15]. Another issue, though it is not a real portability trouble, is represented by the fact that sometimes the events do not work as we expect [MSHN17]. This problem not only might lead to an incorrect analysis but also represents a source of divergence among the several vendor solutions. A standard is highly required because the increasingly complexity of the PMUs, as well as the number of capabilities these provide, may result in a dog chasing its own tail. Software should guarantee a compatibility level at its bare minimum.

2.3 State of the Art

In the rest of this chapter, an evaluation of some of the most popular general-purpose profiling tools is provided.

2.3.1 Perfmon2

Perfmon2 is a performance monitoring interface for Linux [JJN08] [Era08], initially developed for the Itanium architecture [SA00] and later extended to all modern processor families. This interface relies on a kernel side which is in charge of interacting with the PMUs and defining some advanced mechanisms that would not be easy (and worthy) at user-level implementation. As a matter of fact, most architectures allow communication with monitor units only at the highest privilege level (ring 0) which only hosts kernel activity. Furthermore, notification systems such as the interrupt-based one result completely compliant with this design choice. Although it aims to provide an uniform functionality set for all the architectures, the measurement precision is limited by the hardware capability rather than the software implementation. Obviously, this is a common issue to all the hardware-based solutions. As seen in the section 2.2, generally speaking, the hardware supports come along with a set of registers decoupled in *configuration* registers and *data* registers. Both are logically mapped by the interface so that a homogeneous view can be provided independently from the hardware details. Compared to other tools, the access point is not stated by a device file, but it is implemented by system calls which give a more flexible approach for checking argument number and types than *ioctl* file operation. The PMU register access results to be direct through those calls. Moreover, to lower the cost of invoking the system calls several times for accessing multiple registers, a single system call can operate on a set of registers in one-shot. Perfmon2 works using contexts that can be of the following types:

- *per-thread*: this context allows the profiling of individual threads execution exploiting some nested operations in the context switch function (e.g., PMU state saving and loading)
- *system-wide*: this context spreads the analysis activity on the entire system (under some limitation).

The kind of a context is defined at creation-time and a *file-descriptor*, returned after a context creation syscall (`pfm_create_context`), represents the related *id* for further operations. Per-thread and system-wide contexts are mutually exclusive. Thus a context cannot provide dual analysis. A context extends it activity after an `exec` syscall, but it does not persist on

child threads born from a `fork` or `pthread_create` invocation. Directly attaching the child process to the context does not always make sense and can be easily done in a second moment by the child itself. Moreover, a thread can be attached or detached from a context at runtime. Unfortunately, a system-wide context can work only on a CPU at a time. As a consequence, all the threads that want to be subject to that analysis must be bound to that CPU too. Furthermore, the controlling thread accessing the PMU data registers, because of the Perfmon2 implementation, must run on the CPU of interest to successfully complete the request. In order to ease the user-level process of working on several CPU contexts, a library has been released as part of the *libpfm* [Lab] package. Perfmon2 can execute the monitoring activity by means of the hardware counters programmed in *counting* mode, or the *sampling* mode that may use a more advanced technology such as Intel PEBS. Even if the latter method is entirely manageable from the user-level, some kernel support has been provided to mitigate the overhead incurred during sampling action.

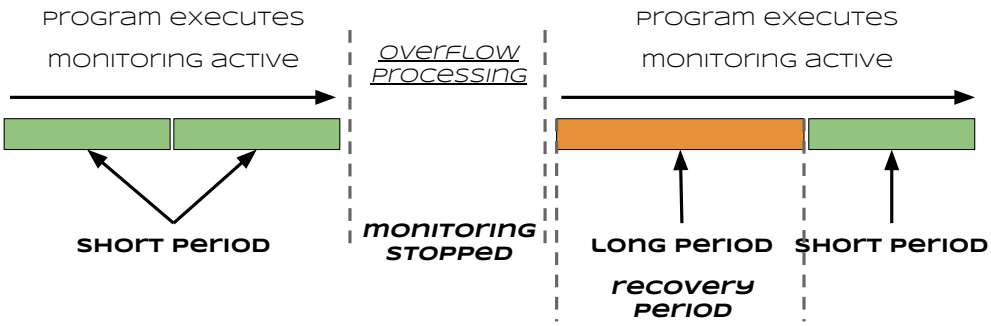

FIGURE 2.3: *perfmon2 short an long values usage [Era08].*

Perfmon2 takes advantage of three different values for managing the sampling period during the profiling activity:

- *value*: represents the first value used. It is usually quite large in order to avoid sampling of preliminary tasks like start-up activity.
- *short*: is used as normal sampling period when the analysis is at steady-state.
- *long*: after counter(s) overflow, some extra actions should take place such as the execution of the notification routine. To hide the perturbation of such a mechanism, a longer (than usual) period is supplied.

The sample collection is performed by using a kernel-memory buffer which is directly mapped in user-space in read-only mode. This overcomes

some performance problems allowing *fast*[12] writes, upon sample generation, without incurring in swapped-out memory handling. Additionally, the cost to access the buffer turns to be far smaller than other ways such as system-call approach.

The interface implementation also provides some precautions to enhance security aspects. Since every user in the system may use the perfmon2 functionalities without the requirement of particular privileges (e.g., being the administrator) it is not possible to ensure a well-behaved monitoring activity. An user can retrieve information about the activity of specific thread only if he or she has the permission to do that. Generally speaking, access to a thread data can be successfully performed only if it is possible to *signal* that thread.

2.3.2 OProfile

OProfile is a system-wide statistical profiler for Linux able to perform the code execution analysis with low overhead [OPr17] [Coh04]. It provides a large support for performance monitor units on a variety of architectures and is compatible with most of Linux distributions.
Its activity can be used to monitor processes either at thread granularity, thus specifying a process or a set of threads to observe, or at the system level, watching the entire system progress including interrupt handlers, shared libraries and kernel modules.

Oprfiler design can be divided into three different blocks: kernel *driver*, user-space *data collection*, user-space *data processing*. The figure 2.4 depicts a schema of how these components are linked together.

The kernel driver represents the central part where the profiling activity takes place. Upon support activation, the underlying hardware registers are saved into an old state variable then configured[13]. In the setup stage, along with the counters programming, some extra elements may need to be configured such as the APIC in order to enable interruption mechanisms for sampling notification. The driver provides a per-CPU buffer which will be used to contain the data samples. Because the sample insertions may occur in NMI context, it is not possible to adopt locking synchronization mechanisms. Consequently, it is adopted a lock-free implementation, which takes advantage of two indices for reading (*tail*) and writing (*head*) samples. This small buffer is later accessed by the user-space daemon that reads its content and populates a bigger buffer known as *event buffer*. This is used

[12]Kernel memory cannot be swapped out, so it is always available when requested.

[13]Saving the old values allows the driver to restore the last context once the support is shut down. This can result useful in a situation such as the NMI watchdog takes advantage of performance counters.

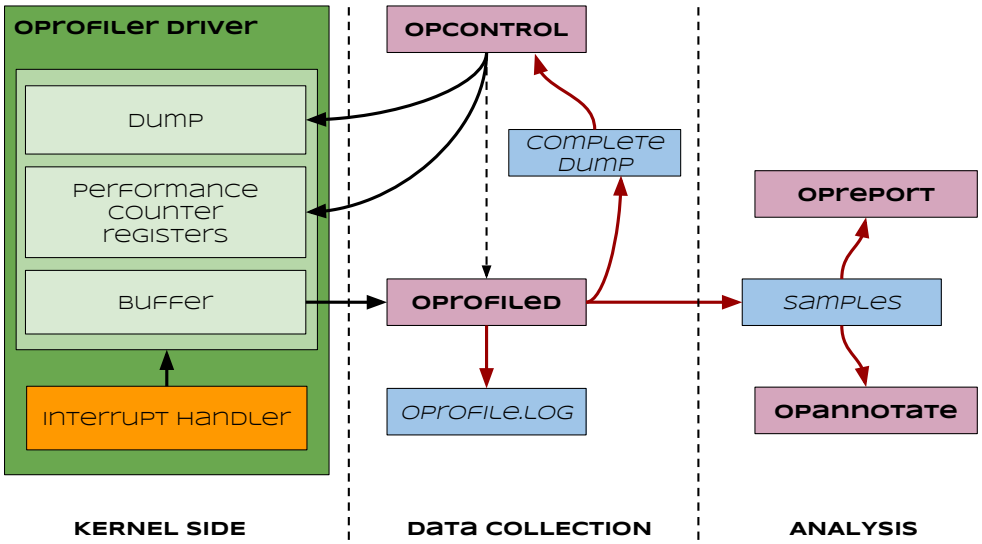

FIGURE 2.4: *OProfile general schema.*

as the bridge between the kernel monitoring activity and the final analysis stage. Furthermore, the per-CPU buffers adopt an *overwriting* policy such that when a new data element is ready, it is discarded if there is no space left in the buffer. A handy ability, especially when working in sampling mode, is the throttling control. In fact, during monitoring activity, the sampling rate may be too high and produce a consistent slowdown. Even if throttling events are notified to the user through console messages, there is no automatic sampling period adjustment which is demanded to the user action. Oprofile can be split into a set of tools and command that regulate the interaction with the profiler, each one designed for specific purpose [Cor16]:

- *operf* and *ocount*: these are profiling tools. While the former bases its activity on the perf_events subsystem provided by the Linux kernel, the latter is used to count some hardware-event occurrences. Both can be used to monitor a single processor the entire system.
- *opannotate*: this command enhances the source or binary code with annotation obtained from sampled data.
- *oparchive*: this utility clusters profiled data and executable in a specific directory for further analysis.
- *opcontrol*: this is the old method to manage OProfile. Now it is deprecated in favor of the operf.

The figure 2.4 depicts a high-level view of the oprofile structure also illustrating the different phases during a profiling session.

2.3.3 Perf Events

Perf, also known as *perf_ events*, *Linux perf events* or *perf tools*, represents the most advanced analysis suite already built in the Linux distributions [Gre17] [Gho16]. Initially, its name stood for Performance Counters for Linux (PCL), and it was created with the purpose of accessing the functionalities provided by the available hardware units. However, as the time went on, its scope has been extended by adding more analysis capabilities such as software events observation and tracing support. At present, perf supplies a large collection of events from both software and hardware side. Although the hardware information comes from on-chip performance counters, the software events are provided by the Linux *kprobe* and *uprobe* debugging utilities. Because of this wide range, some other solutions, like perfmon2, have been abandoned in favor of perf, while APIs, like PAPI, now take advantage of its facilities. As consequence of the fact that perf is not only intended for PMU access and use, studying all its components may result extremely time expensive besides the fact that those are not directly related to this work.

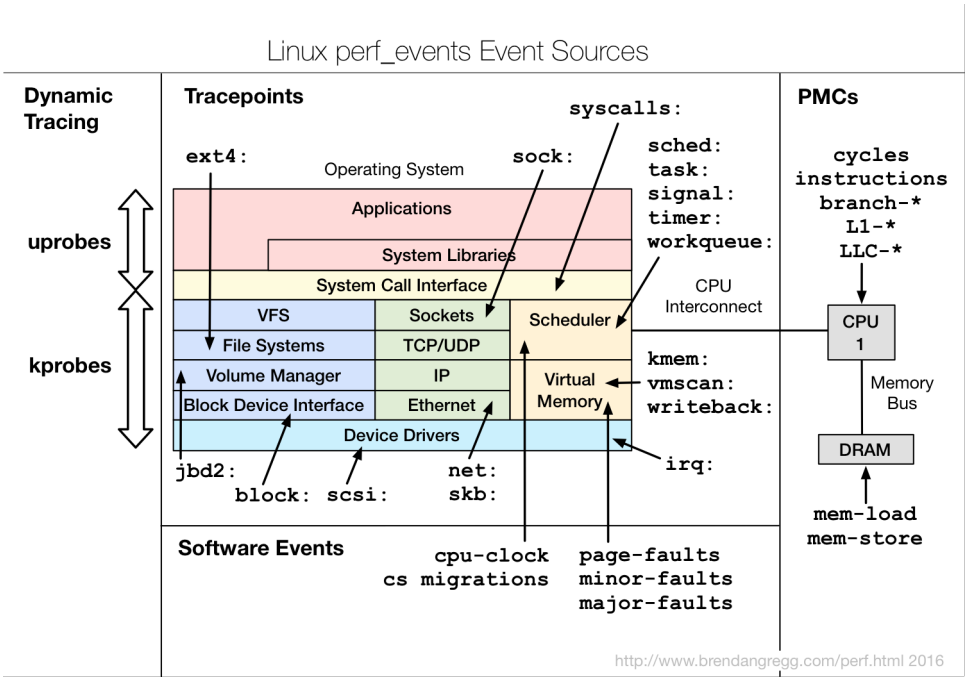

FIGURE 2.5: *Perf structure map [Gre17].*

When attempting to study perf architecture, the first obstacle is the

insufficient documentation apart from some useful comments within the kernel source code. In this section, we try to describe the perf working principles connected to the only hardware profiling activity.

The perf core is shipped within the Linux kernel and includes the *mini* drivers needed to configure, enable and stop the PMU activity. It is important to state that the primary perf target is not an uniform PMU interface to work with, but maximizing the usage of the available hardware support. Even though perf_events is able to work with several hardware supports, the software interface is biased and *event-oriented* rather than *instruction-oriented*. Some supports instruction-based like IBS are forced to be used through this interface, resulting in an events remapping. Along with the perf_events, the system is equipped with a user-space tool, simply *perf*, that allows interacting with its kernel counterpart. It provides the useful *perf list* command which returns the list of all events (also highlighting if the event is a hardware or a software type) that can be observed. If the present hardware counters cannot work with some event, *cache L1 hit* for instance, that will not be reported.

Even though the user-space tool represent a *ready-to-use* solution which helps users carry out either advanced or straightforward evaluation sessions, there is no employment of an user-space daemon for retrieving collected data. Thus a user program can conduct the entire analysis by itself with a *direct access* to perf core. This point of access is given by the `perf_event_open` system call [Man17]. By studying its parameters, it is possible to look at several implementation choices.

A single `perf_event_open` invocation is intended for a single event activation, and further events require additional calls. Perf can conduct its analysis by following the activity of a single process/thread[14] or even the entire system. Furthermore, it is possible to bound the event collection to specific CPU such that different groups of events may be observed on different CPUs. Starting a system-wide profiling operation requires the root permission as well as registering a thread analysis by another thread requires the right credentials. Upon `perf_event_open` completion, a file descriptor is returned and then used to obtain the generated data. Perf automatically understands how to perform the required event profiling, and does not expect the user to have any knowledge of the underlying hardware platform. For instance, if a measurement requires a specific precision level (skid degree), only sophisticated support like Intel PEBS and AMD IBS are taken into account because these are the unique mean to accomplish this request.

[14]The user-space perf application allows supplying a set of thread/process IDs for a single evaluation. This is translated into a series of *perf_ event_ open* calls, one for each thread/process.

However, besides these *general* events, the user may ask for more specific events that cannot be associated to any of the available options. To cope with this issue it is possible to specify the *event code* as provided from the vendor manuals [Dev10b] [Cor17].

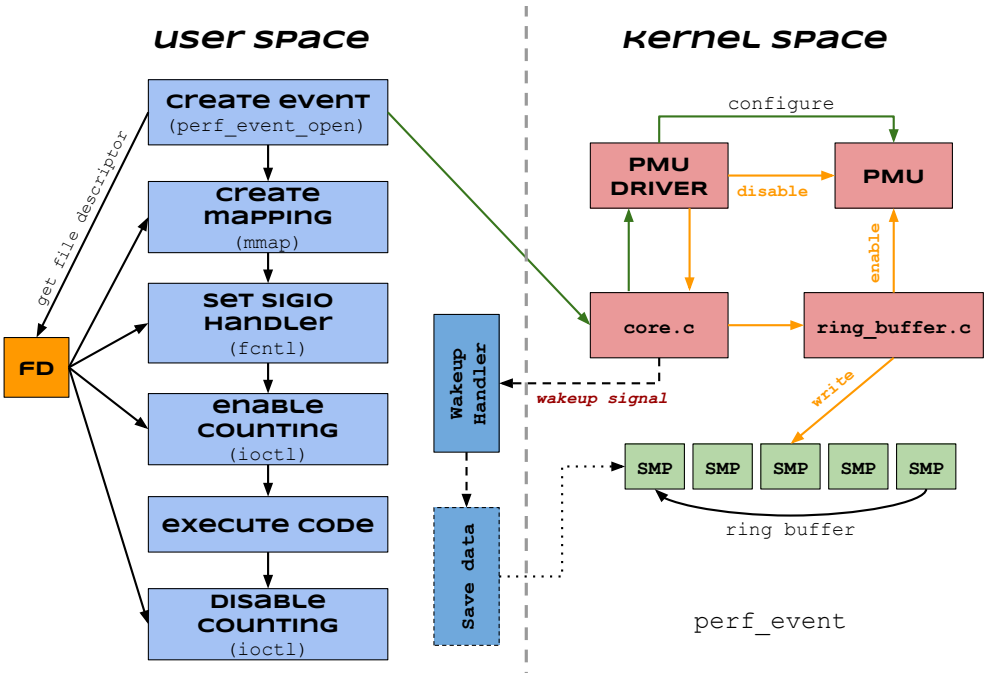

FIGURE 2.6: *Perf simplified control flow for sampling mode analysis [Gho16].*

Perf handles either *counting* or *sampling* events. Overall, the two types of events follow the rules described in the previous sections, but, according to the event type, perf may adopt different structural techniques. As a matter of fact, in the former case, the result is just a value (aggregation value) and can be retrieved via a *read* system call on the event associated file descriptor. The sampling case is managed differently because the size of the data is determined by the number of sampling elements generated during the profiling activity. Every time a sample is generated, it does not matter where it comes from (counter overflow or advanced support), it is saved in a *ring buffer*. This buffer can be directly mapped in the user-space memory region through a `mmap` system call[15]. Indeed, perf's job is samples gathering, and the buffer needs to be empty before incurring in sample overwriting. The reading process can be registered to a notify mechanism such that it is always informed via a *wakeup* signal whenever the buffer is

[15]The memory size should be of the form $1 + 2N$, where N is the number of pages (generally a page is 4Kb) and *1* refers to a metadata page.

about to be full. The way to perform this registration can be through a `select`, `poll` or `fcntl` calls on the file descriptor of interest. While the first twos put the process into a *waiting* state until the reading conditions are met, the third requires the implementation of the signal handler for the management of the `SIGIO` signal.

Although some supports like AMD instruction-based Sampling provide the possibility to define the sampling period by the number of instruction retired, perf considers only the time domain. Perf is extraordinarily flexible and can enhance the event analysis in several ways. Besides the ability to extends the ongoing profiling activity on child threads, it can create powerful event groups. This allows treating grouped events as a unit which benefits from context execution. Moreover, it is possible to discriminate the kernel-side activity so that it will not be part of the analysis.

2.3.4 Comparison

We cannot directly make a comparison among the above-described tools because those do not turn out to be direct competitors, but represent the set of elements that defined the evolution of the profiling tools on Linux systems. OProfile — which has been used for many years as principal profiling tool by Linux users — can be considered as the predecessor of perf (both kernel and user parts), while perfmon2, more appropriately, represents an interface for accessing the PMU as well as perfctr. Both defined the base APIs such as PAPI before perf_event coming.

Primary interfaces have been designed with a specific purpose or tool in mind and, most of the times provided orthogonal features. For instance, early versions of OProfile allowed only system-wide analysis, making a simple activity like specific thread retired instructions counting unavailable. On the other hand, perfmon2 — which represents a finer tool for system-wide and per-thread profiling — did not provide any automatic means for data retrieval — such as the OProfile daemon — and the application should be given with an own method to access PMUs content.

Even though OProfile takes advantage of a device file for retrieving data from profiling analysis, perfmon2 does not seem to have direct support for saving gathered data. As a consequence, there must be a controlling thread accessing the PMUs state which is required to be running on the CPU hosting the PMU of interest. A user-space library helps the user to accomplish such a strategy.

Perf_events comes up from the need for a standard interface to access

the PMCs functionalities. The new project is built upon the perfmon2 fundamentals which aimed at a direct implementation in the Linux kernel to overcome approaches such as module loading and kernel patching adopted by OProfile. Perfomon2 initially provided the system with as many as 12 system calls, that, after some improvement and code redesigning, were reduced to 4. On the contrary, perf_events provides just a system call to create and manage a whole monitoring session for a specified event. This tool has been broadly appreciated by the community which made it be the principal analysis tool on Linux. Its support to PMUs has been further improved by providing software event inspection along with tracing capabilities provided by additional tools such as kprobe and uprobe. We cannot directly compare OProfile with perf. As a matter of fact, the former is strictly a profiler tool while the latter is definitely something more.

The number of events that is possible to observe on several architectures represents a big problem which makes a sophisticated system like perf_events take some compromises. Kernel-side perf, named perf_events, is shipped with a user-space tool, simply perf, which allows starting profiling session and analysing information offline. Decoding of such data is an essential operation with the purpose of associating a useful meaning to the gathered information. Although the perf_events features may be explicitly used by an application, the developer should know the complex hierarchy of the perf data structures as well as how to interpret them.

Our work comes into the world from the dual need of exploring the potentiality of the most advanced PMU supports — such as PEBS and IBS — and conducting such a study on machines at our disposal. In accordance with our machines, which are equipped with AMD Opteron processors, we decided to start investigating the IBS technology capabilities. We wanted to evaluated the such a support on its performance impact as well as its accuracy. By taking advantage of other tool to access PMUs, we would not be able to obtain a direct control of the hardware. Furthermore, analysing specific questions such as the influence of the NMI handler routine on the execution efficiency would be impossible because those logics are internal parts of the tool implementation. Finally, although it is not compliant with the advanced functionalities perf provides, it grants a better focusing on hardware support that turns out to be effortlessly configurable and easily adaptable for our other research purposes.

CHAPTER 3

The HOP Kernel Module

Don't lower your expectations to meet your performance. Raise your level of performance to meet your expectations.
— Ralph Marston

The *Hardware-based Online Profiler* (HOP) is a kernel module for Linux x86-64 based on Instruction-based Sampling (IBS) technology by Advanced Micro Devices. It aims to provide statistics of running applications by profiling their behavior in the system. Unlike most profilers, it provides information in on-line fashion at low overhead so that an *on demand* tuning of the application being observed can be applied according to the execution flow. HOP is a pure-hardware profiler and its engine is powered by the underlying architecture. The version presented in this thesis takes advantage of AMD Instruction-based Sampling support. This statistical approach allows collecting data in form of samples whose presence is notified via system interrupts. Keeping the overhead as low as possible is one of the HOP priorities, and this was the reason for a direct interface to the hardware elements. Many software applications, due to the problematic way of interfacing to the PMUs, make use of the facility provided by third-party codes like external libraries. However, this method does not give the possibility of a full control on hardware, increasing the code dependability and sometimes resulting in a disappointing solution.
HOP directly works on model-specific registers (MSRs) and this naturally requires a broad knowledge of the hardware architecture, which most of the time is specific for each processor model. The required ad-hoc code to make the profiler portable on several architectures has been the primary reason to limit the actual HOP compatibility to the AMD Family 10Th architecture only. Although Linux is an operating system not directly designed to support multi-threads activity, the increasing number of SMP machines made its developers adopt some solutions. Briefly, in modern Linux versions we can find *processes* and *threads*[1]. The former group is used to wrap kernel

[1]Some authors refer to them as *lightweight processes*.

and user tasks which can be represented by an application, a service and so on. The latter instead, concerns the use of multi-threading execution to concurrently carry out one or more sub-jobs[2] within the original process task. Having more logic units lets programs exploit the parallelism given by multicore architecture since each thread can be concurrently executed on a different computing unit. HOP is created to observe the activity of each process meticulously by working at thread-granularity. This ability is compelling because it lets the user analyse particular components of the processes by focusing on specific threads of the program. Furthermore, it is possible to perform a combined profiling activity on more processes at the same time.

3.1 Management

When HOP is loaded to be part of the active Linux modules, it makes an underlying architecture support audit.

1. First of all, it checks whether the current operating system is running on an AMD Fam. 10Th processor(s) and consequently the Instruction-based Sampling support availability. This step is based on a series of CPUID queries[3] [Dev11] executed in order to identify processor *vendor* (AMD required), *family* (FAM10th required), *IBS supports* (IBS_OP required) and some extra features like *Branch target support*[4].

2. After the hardware check, the next phase is the system setup. The section 3.2 will explain in detail how this step is performed. As a little preview, the IBS support has to be activated, and the system must be informed of its presence. The Local APIC within each core is configured for handling the IBS samples generation, and the Linux system is equipped with a new interrupt handler that is in charge of collecting the records upon their presentation.

3. Finally, the hardware and the system are ready such that HOP can proceed to allocate its internal structures.

The stage we are going to present in the rest of this section represents the hearth of the management system in HOP. HOP exposes a special control

[2] A good parallel code must take care of the synchronization activity that manages the job performed by each thread solving the possible conflicts arising in the critical sections.

[3] CPUID stands for CPU IDentification, and it has been designed to allow software to identify details of processor.

[4] This technique is a primary form of portability check. In fact, it is possible to design the software to work with the available supports without considering missing capabilities.

device, placed at */dev/hop/*, named *ctl*. The user uses that as unique point of interaction with the module by performing a set of *ioctl commands*[5]. The most important operation is the *thread registration* that inserts a new Thread ID (TID) into the module logic. This registration can be performed either by the thread itself or in another element such as a pairing thread or a direct user action. Internally, there is a special hash table which keeps track of the already registered threads and it is intended for a quick lookup for further operations. Upon thread registration, a new metadata structure called `pt_info` (profiled thread information) is built and inserted into the module hash table. pt_info provides several information, which will be described in the next sections, concerning the thread management and related components among which:

- `tid`: thread ID.
- `mn`: minor number of the related character device.
- `ctl`: reference to the last stack entry.
- `dbuf`: reference to the `pt_dbuf` (profiled thread dedicated buffer).
- analysis variables: set of variables holding extra information concerning the profiling activity.

Listing 3.1 provides the complete structure.

```
1  /* profiled thread information */
2  struct pt_info {
3    int tid;
4    struct minor *mn;
5    unsigned long *ctl;
6    struct pt_dbuf *dbuf; /* sample buffer */
7    volatile unsigned long busy;
8    volatile unsigned long kernel;
9    volatile unsigned long memory;
10   volatile unsigned long samples;
11   volatile unsigned long overwritten;
12   wait_queue_head_t readq;/* used for poll fop */
13   struct mutex readl;   /* read lock */
14   struct cdev cdev;   /* cdev for char dev */
15   struct hlist_node node; /* hashtable struct */
16 };
```

Moreover, the module makes a character device per each thread such that the user can access it and read the gathered data. However, the maximum number of available devices is limited to *255* thus limiting the number of profiled thread concurrently registered as a consequence.

Even though the hash table provides a fast way to look for registered threads, it is not used during profiling activity to discriminate analysis actions. We adopted an alternative approach which links all the data related

[5]IOCTL is a special system call that lets the user manipulate the device state through a set of custom operations.

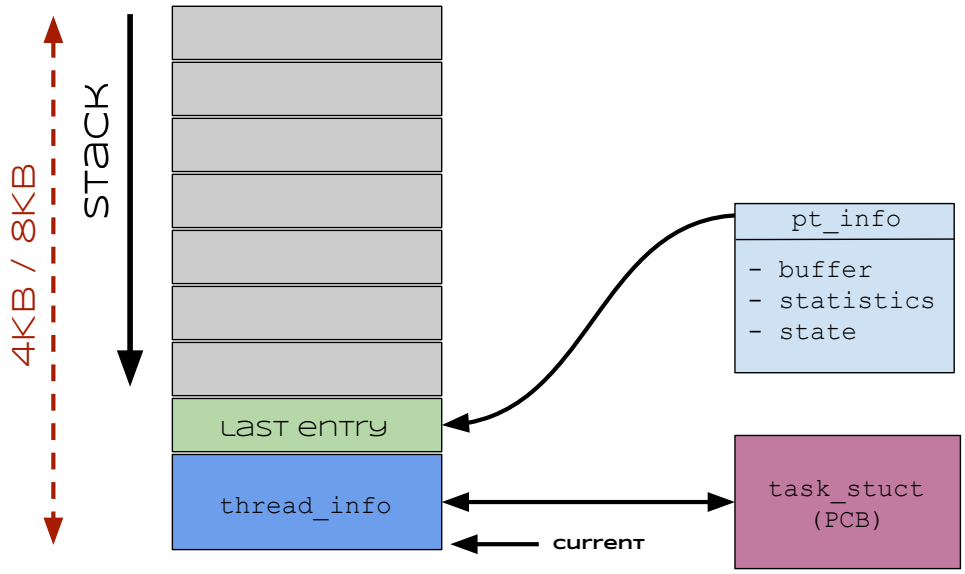

FIGURE 3.1: *Stack patching schema*

to a profiled thread to its kernel stack. The figure 3.1 helps to understand this operation. At any time, the process control block (PCB) of the currently scheduled thread can be accessed via the `current` macro available in Linux. The PCB contains a reference to the kernel stack associated with the process and can be always obtained in a constant time[6] when the process is scheduled. This approach lets the pt_info structure potentially be accessed in a very fast way because it just requires the traversal of a few memory pointers. When an interrupt arises for a new sample creation, the handler can easily access the current thread stack since it is being nested into the currently scheduled thread which is also the owner of the sample (see 3.2). Thus, the last kernel entry contains the address pointing to the profiled thread info structure which provides the information about the related thread state, collected statistics and buffer. When the buffer is requested to be read, its position is retrieved by scanning the hash table instead, making all the process logically simpler. The registration counterpart is the unregistering of a thread which disables analysis activity, waits for pending writes, and eventually cleans up the thread metadata removing its entry in the hash table.
The profiling activity can be globally controlled by enabling/disabling the monitoring at the module level. This directly affects the monitor function that starts IBS at runtime (more details in the section 3.3). Additionally, we designed another control mechanism that directly handles the thread exercise. In fact, it is possible to define the profiling policy for each register

[6]Compared to searching in a generic data structure.

thread such that some can be profiled and other just ignored by the IBS support although they are still registered. This capability turns out to be very useful in complex studies that require disabling and enabling thread activity more time per execution.

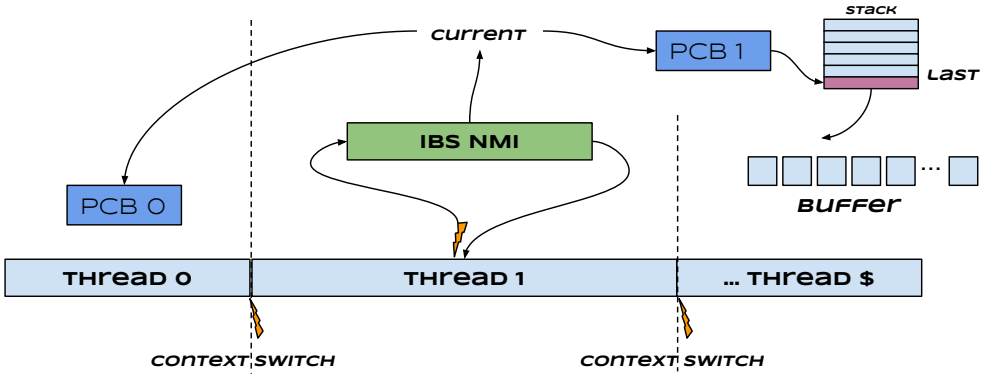

FIGURE 3.2: *NMI handler access to profiled thread buffer exploiting stack patching solution.*

HOP is a flexible solution that tries to adapt to several contexts. This is enforced by the ability to modify at runtime some parameters as the sampling rate and the buffer size. While the latter is not retroactive and affects only the subsequent thread registrations, the former is instantaneously updated such that a malleable activity can be executed. This competence lets the profiler be tuned according to the working load and contextually leverage the overhead in extreme cases.
The last but not the least, the thread devices can be accessed both by a read and ioctl system calls. The former, as already introduced, is used to obtain the profiled data while a set of aggregated statistics, such as the number of generated samples, can be received concurrently by the latter. All these features aim to make the HOP interface very simple so that a smooth integration is guaranteed. Table 3.1 provides available commands along with a brief description.

3.2 The IBS Setup

The first task the module executes is checking the compatibility of the underlying platform against IBS capability, understanding which features are available for the given support (e.g. branch prediction support) and, ultimately, it setups the environment making all the elements ready to be used. Even though the support has been limited to AMD Fam. 10Th architecture, conceptually, the operating principle is shared among all AMD archi-

Command	Description	Action
PROFILER_ON	globally turns on the profiling activity	CTL
PROFILER_OFF	globally turns off the profiling activity	CTL
CLEAN_TIDS	removes all the registered thread and resets the sampling information	CTL
TID_STATS	retrieves aggregated information such as number of generated sample or kernel mode	THD
ADD_TID	adds a new thread to be profiled. When the action completes a new thread device is available	CTL
DEL_TID	deletes a registered thread from the module. All its data will not be accessible	CTL
START_TID	enables the profling activity for the specified thread	CTL
STOP_TID	disables the profling activity for the specified thread	CTL
SET_BUF_SIZE	sets a new per-thread buffer size	CTL
SET_SAMPLING	sets a new sampling frequency value	CTL
READ	reads the collected samples	THD

TABLE 3.1: *Main functionalities provide by HOP. The action column refers the target of the command:* CTL *indicates the control device and* THD *refers to the thread device*

tectures that support IBS. Of course, several improvements and extensions of the capabilities took place during years and, hopefully, new models provide a far better support than AMD Fam. 10Th, which represents the first architectural solution with IBS. The more exciting part of this preliminary phase, without a doubt, was represented by the IBS support configuration. Notwithstanding, before going deeper into details, it is worth providing a little overview of the components which play a role in this stage.

The Advanced Programmable Interrupt Controller

The *Advanced Programmable Interrupt Controller* (also called Local APIC), within any individual core, is an essential element used to manage the interrupt redirection and interrupt exchanging among processors.

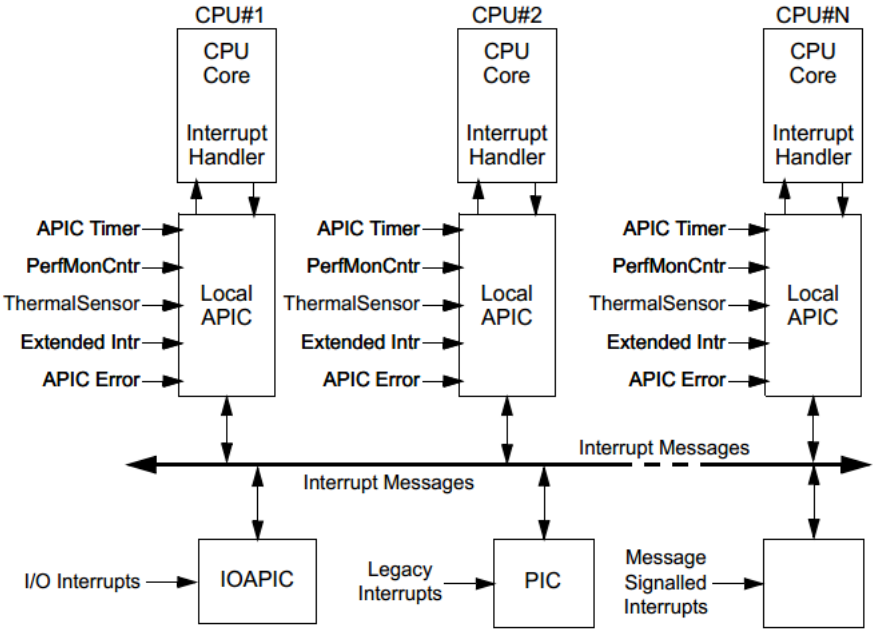

FIGURE 3.3: *Block Diagram of a Typical APIC Implementation [Dev17]*

It has been designed to overcome the limitation of the older PIC in the SMP systems. Along with the per-core Local APICs, the architecture provides an extra element called IO-APIC. It is designed for the interrupt management within a multi-processor system and acts as a dispatcher for interrupts coming from out-of-core sources. It can redirect interrupts in both dynamic and static fashion, thus managing the distribution across all CPUs. The IO-APIC does not directly present the interrupt to the target processor, but, exploiting the presence of the APIC bus which interconnects all the APIC elements, redirects that interrupt to the related Local APIC.

When something wants to cause an interrupt to a core, it will send a request to that core's APIC, which handles interrupting the core and sending it to the correct interrupt handler for that event. Taking into account the AMD Family 10Th architecture, there are several sources of possible interrupts that can be sent into a CPU core's APIC:

- *I/O interrupts*: interrupts coming from I/O devices or redirected from the IO-APIC.
- *Legacy Interrupts*: legacy interrupts managed by the PIC and redirected to Local APIC.
- *Interprocessor (IPI)*: interprocessor interrupts.
- *APIC Timer*: interrupts coming from the programmed APIC timer.
- *Performance Monitor Counter*: interrupts from the performance monitoring counters.
- *Thermal Sensor*: interrupts from internal thermal sensors.
- *Extended Interrupt*: programmable interrupts (see below).
- *APIC Internal Error*: interrupts caused by an error detection within Local APIC.

The figure 3.3 depicts the scheme of a typical APIC implementation. Although one of those sources is *Performance Monitor Counter*, it shall not be considered valid for IBS, but rather for traditional performance counters. IBS, instead, exploits the "extended interrupts" mechanism. *Extended interrupts* is a method used by AMD to *extend* the local interrupts by adding more interrupt sources. This is made possible by adding some extra entries into the Local Vector Table (LVT), which can be configured to handle new interrupt types [Dev10b]. Precisely, by exploiting this mechanism, each time an IBS sample is ready and signalled by the core, it tells the APIC to cause an extended interrupt for type *IBS Sample*. The APIC performs this operation by looking up the LVT where a particular interrupt number should redirect the processor. The terminology used here is that we are *vectoring the processor* to the handler location. It is possible to configure the entry associated to IBS for generating a specific kind of interrupt, which can be one among

1. *Normal* Interrupt, used to signal the CPU that *something* at hardware or software level happened during code execution,
2. *System Management* Interrupt (SMI), used to manage firmware-level functionalities such as power management, and

3. *Non-Maskable* Interrupt (NMI) that, as the name suggests, cannot be masked, thus it is always presented to the CPU.

The last type is the one used in this module implementation. The reasons for this choice are discussed in the section 3.4. Generally speaking, this kind of interrupt is used to signal critical event like a hardware fault. The way an NMI is associated with its handler is entirely different from the technique used for the normal interrupts. On each x86 core, there is only a single non-maskable interrupt handler[7]. When performing the registration of a new NMI handler, actually, the system put it into a chain that is walked through every time an NMI is generated. In that case, all the handlers will get called until one will *handle* the interrupt. How can is it possible to discriminate the right handler among all?
The handler routine shall bear the audit part, which queries specific registers (or memory locations) and looks for particular conditions that may generate its event, revealing if the interrupt was intended for it or if this NMI came from someone else. For instance, the IBS handler reads a defined MSR and checks whether *valid sample bit* is set.
Indeed, the NMI management is far complicated [Ros12] and some conditions could arise such that when an NMI is signalled, more than one handler may concurrently execute. However, most of this job is directly done by the Linux operating system, while the only responsibility left to the programmer is the right implementation of the handler routine.
The NMIs provide an excellent flexibility because it is possible to watch *what is happening inside* normal interrupt handlers, and so to take related IBS samples. Only NMIs can interrupt regular interrupt handlers [8]. Nonetheless, it is possible to set up the support on a specific sub-set of available cores, such that more adaptive policies can be applied.

3.3 The Schedule Hook

One of the problems of the AMD Instruction-based Sampling implementation is the lack of conducting the sampling activity on specific threads in the system. Moreover, this shortfall affects the overall system performance since the IBS support extends its activity not only on system threads, but also on kernel code nested during the execution of a user-space code. In fact, during such an execution some events may occur—for instance the system tick interrupt which might trigger the system context switch—and their handling is performed over the current thread context. From the user and

[7] All NMIs go into interrupt handler number 2.

[8] Actually, an interrupt priority-based policy allows higher-priority interrupts to stop lower-ones.

system point of view, such an event management and related code execution is nested into the thread activity. This job may not be directly related to the current thread work, but to other element like operating system housekeeping. Furthermore, code instructions concerning those tasks executed during IBS support analysis will be part of the sample collection leading in some cases to a resource waste because kernel investigation is not always relevant for final analysis. The simplest solution to cope with this lack is the filtering out of undesired samples at either collection or analysis time. Although it is not a system-intrusive solution because does not need any particular system capability or dependence, it cannot discriminate the sampling on determinate threads nor save time to perform the profiling only on the wanted code domain (kernel or user). However, we decided to face this problem in an advanced fashion by directly operating on the schedule function within the Linux kernel. The general idea is inserting extra code to be executed at the end of the function, such that, after each context switch, it is possible to know which thread is currently scheduled. Based on this concept, the module can activate or deactivate the IBS sampling, which, potentially[9], will monitor only the profiled thread activity. Therefore, the sampling even works on the kernel side code executed during the profiled thread performing. HOP exposes a parameter that represents the preference to keep kernel sample (samples are tagged with a kernel flag) or discard them inside the NMI handler function, thus saving the overhead of reading all IBS data from registers. Coming back to the scheduling hooking process, we thought several ways to perform it. A compiled custom version of the kernel could be a solution, however, from the point of view of the final user, it is too invasive and complicated to be adopted. We decided to embrace, instead, a different solution, which acts by *dynamically* patching the schedule function at runtime [PQ17]. The authors introduce this module that, by inspecting the system-map[10], rewrites parts of the executable of the kernel upon being loaded. The patching operation inserts at the end of the schedule routine a flow variation such that the control is given to a custom function, defined by the authors as *schedule-hook*. According to the patching schema, the original schedule function will never reach the *return* instruction, which instead, will be executed by schedule-hook as the natural continuation of the function. It is important to highlight that the schedule-hook code comes just after the context switch is finalized, thus the action is performed in the new thread context. The power of this solution lies in its nature of being a module, that, once loaded, provides the possibility to hook any function

[9]During the thread execution, other code may be nested to be executed like kernel housekeeping, or interrupt routines.

[10]The system-map, generally places into /boot, represents a symbol table used by the kernel which provides an association between symbols names and memory addresses.

to be executed in place of schedule-hook. Moreover, it perfectly matches our goal. The algorithm 3.1 shows the `thread_monitor_hook` function in charge of enabling and disabling IBS during module activity. Its routine is straightforward and is kept as light as possible. As the first step, it checks the global profiler state. If it is disabled, IBS is not required, and the support is turned off (`1-2`). Otherwise, according to the stack patching solution described above, the last entry of the current thread stack is inspected, and a quality audit is performed (`4`). In particular, the CRC code allows filtering out threads for which its content is not consistent, while the per-thread `ENABLED` bit identifies whether or not the profiling analysis is required for that thread. The enableIBS and disableIBS methods do not work on the MSRs each time their are invoked. That would be very expensive since this function is invoked at a very high rate. Instead, a logical state—tied to the per-core IBS structure—is given such that it is possible to make a fast check before acting on registers. For instance, if the support is disabled, the scheduling of a *not profiled* thread would invoke the disableIBS method which results in a useless operation. Our technique catches such situations.

Algorithm 3.1 Thread monitor hook

```
1: if profilerState = disabled then
2:     disableIBS()
3: else
4:     if validCRC(current) ∧ isEnabled(current) then
5:         enableIBS()
6:     else
7:         disableIBS()
8:     end if
9: end if
```

3.4 NMI Handler

The NMI handler represents the hearth of the module profiling activity. As anticipated in the previous section, in the Linux system an NMI handler routine is registered in a list (chain) and potentially invoked for each NMI that is generated. The routine must investigate the reason of its invocation by checking if the conditions of the event for which is designed match the actual state. The main problem to deal with when designing an NMI handler, even more important than a regular interrupt one, is its *complexity*. An NMI runs in a very particular context which stops the execution of the other code, no matter of which task was executing. Moreover it enjoys the ability of *not being interrupted* which means that no one can block its

execution[11]. If the routine takes too much time to complete, it will cause the delay of all planned activities. Additionally, a routine stall due to any reason turns out to be a system hang. In our case the IBS NMIs occur very frequently, thousands or even millions per second, representing the primary source of overhead. The less the time spent in processing the interrupt handler routine, the more the time assigned for the thread task execution. The handler function is shown in the 3.2 algorithm. Basically, it makes several fast tests before performing the IBS sample collecting.

Algorithm 3.2 NMI handler

```
1: retval ← NMI_DONE
2: msr ← RDMSR(IBSOPCTL)
3: if not VALIDSAMPLE(msr) then
4:     goto OUT
5: end if
6: retval ← NMI_HANDLED
7: if not ISACTIVE(LOGICIBS) then
8:     goto OUT
9: end if
10: if not (VALIDCRC(current) ∧ ISENABLED(current)) then
11:     goto OUT
12: end if
13: if TESTANDSET(current.PROCESSBIT) then
14:     goto OUT
15: end if
16: if not KERNELSAMPLING( ) ∧ CPL( ) ≠ 3 then
17:     goto SKIP
18: end if
19: sample ← READALLMSRS()
20: ADDEXTRAINFO(sample)
21: INSERT(current.BUFFER, sample)
22: skip :
23: RANDOMIZEANDENABLEIBS()
24: CLEAR(current.PROCESSBIT)
25: out :
26: return retval
```

According to the Linux policy, a generic NMI handler must tell, at the end of its execution, if the interrupt was intended for it or someone else. To

[11]In particular cases such as the generation of an exception during the NMI execution, it can be stopped to handle that exception. This advanced mechanism is part of the Linux system logic [Ros12].

indicate that, the handler returns a 0 value (`NMI_DONE`) if it could not manage the event, or a positive value (potentially `NMI_HANDLED (1)`) whether it handled the interrupt[12].
Even though IBS provides two different sampling support types, fetch and op, our module takes advantage only from the latter which produces data related to code execution. Examining if the interrupt is an IBS type is relatively simple and is done through a query on define `IBS_OP_CTL` MSR [Dev11]. This register holds all the control information related to IBS Execution activity among which the `IBS_OP_VAL` bit that tells if a valid sample is ready to be read (`1-5` in 3.2). Every time an IBS sample is generated, that bit is set and, consequently, an interrupt is fired[13].
Once this step is over, the handler function is sure that the NMI was produced for it and can carry on its job. At this point the return flag can be set to `HANDLED` so that the system NMI handler is informed to stop running other routines because the interrupt was already handled. A series of extra checks are required to save in the best way the retrieved data:

- *Catching of spurious interrupts* (`7-9`): this is a critical audit that allows decoupling the logical state of the IBS system from the hardware one. Turning off the IBS support by writing on the related register is not an action which is executed atomically, so an NMI may occur meanwhile. In order to avoid the gathering of unwanted samples, an IBS logical state is kept by HOP and inspected every time a profiler-related action should be taken. The IBS real state is always consistent with the logical one, except during these small delays.

- *Thread state* (`10-15`): although advanced mechanisms are provided to ensure the cleanest activity, the NMI generation is not deterministic and may occur during the execution on any thread. To cope with this problem each profiled thread keeps status information within the last entry of its kernel stack[14]. In particular, beside the structure that contains the buffer information, two more bits are provided. The `ENABLED` bit identifies that the thread is marked for profiling activity so that IBS can be activated during its execution. The *PROCESSING* bit is a sort of internal consistency guarantee. From the handler point of view, it is used to signal to other parts of the module acting on the

[12] An handler may manage more than one NMI at a time, like in the IBS case. Op and fetch supports are decoupled and they generate interruption independently.

[13] If both fetch and op support were active, the check would be doubled on two MSRs because the handler must catch both sample types. A curious case is presented when during an interrupt generation both fetch and op samples are available and as a consequence, those two are taken at the same handler invocation.

[14] A CRC code is used to discriminate all the system threads such that only registered threads are considered

buffer that this is already in use and cannot be touched. Therefore, this is used to prevent conflicts during a thread check-out. It may happen that during an NMI processing on a core for thread T, another core is performing the log out for T, upon user request. During the *clean up* phase, the memory allocated for the buffer is freed so the handler may work on wrong memory space. As a way of protection, the handler notifies in an atomic manner its activity on the buffer and every cleaning action will be locked until a job termination is sent back. Therefore, in case the handler arrives after the cleaning activity, it will directly jump to the end of the routine by reading a *busy* `PROCESSING` bit.

- *Kernel samples* (`16-18`): the section 3.3 largely describes how an IBS NMI may arise during kernel-space code execution. HOP provides the possibility of either processing the kernel samples as part of the ongoing analysis or directly skipping them and so saving execution time.

- *Sample collection* (`20-21`): once the validity of the new sample is confirmed, the handler proceeds to gather the related data, packing it into a structure. According to AMD Fam10Th architecture specifics [Dev10b], the information can be retrieved by reading up six MSRs, each one giving specific details. Additionally, the sample is enhanced by extra fields such as time stamp counter (TSC) value and sample mode-type (user or kernel space). The strategy used to write the sample into the buffer is meticulously detailed in the following section.

- *Next cycle* (`23-24`): at the end of its activity, the handler must reset the IBS register for a new sampling cycle. Actually, this is done by clearing the new-sample `OP_VAL` bit (see section 2.2.2) and by randomizing the counter value within the control register. The randomization technique is extremely useful because allows going over the determinism of a loop, preventing the sampling of the same instruction.

3.5 Buffering Strategy

HOP has not been specialised for a determined profiling activity yet. Indeed, it can be considered a general-purpose profiler. Because of this, we provided a buffering mechanism for samples recording that aims to provide an idea of how the buffer structure should be implemented to guarantee the best performance. As already discussed, the critical issue of the NMI handler is the *speed* related to its code execution. The meaning of speed does not stop at the size of code, but it also comprises the quality of the code

itself. The NMI routine is the crucial part of the module that is started upon a valid IBS sample generation, and it is in charge of checking some conditions, and, if true, eventually saving the new record in a dedicated buffer. We provide within the module only one level of buffering such that the buffer structure is directly shared among the writer and the possible readers. Considering the Linux system policy, a thread cannot be scheduled on more than one processors at the same time. Therefore two NMIs cannot arise for the generation of a sample related to the same thread. Since the NMI is the only part of the module allowed to write the buffer, we can conclude that there is only one writer at time and we can avoid managing concurrency for multiple writers. A different situation is presented during the reading phase. For each thread registered to the module for the monitoring activity, a character device is built and used as the main point to retrieve the data. The *read operation* that can be performed on that device is mapped on a file read function (exposed by Linux file operations interface). This function represents the unique way to access the collected data by the userspace-level applications. There is no limit for the number of the processes that can access a specific device. Thus the presence of more readers must be managed in the buffering policy. To cope with this matter, we arrange the data structure as a *Single Writer - Multiple Readers* buffer. Moreover, the concurrent accesses among writer and readers, and readers themselves are managed by atomic operations[15] which are implemented in order to guarantee a non-blocking logic. The introduction of atomic operations perfectly matches the need of a light execution, especially for the writer that might not be easy to provide using other synchronisation methods. Before proceeding to the implementation details, it would be worth to have a look at the used atomic operations. The operations mentioned above are the `fetch_and_add` (FAA) and the `compare_and_swap` (CAS). The former belongs to the `fetch_and_*op` family, where `*op` identifies an operation among `add, sub, and, or` and so on. FAA takes as parameters a memory address and value; it first reads the content of the specificated address, then operates on the two operands and eventually update the memory content returning the old value. The operations are executed atomically without the risk that another operation reads or writes the memory address before these end. Conversely, `compare_and_swap` allows *swapping* two different values, but it completes without undertaking any action. The accepted parameters are a memory address and two values. The former represents the memory *actual* value, while the latter the *new* value. Before updating the memory

[15] Generally speaking, this group of operations represent the class of *Read-Modify-Write* operations. These are executed atomically, such that, working on the same value, do not incur into the interleaving of sub-operations. Their support is part of the hardware itself and Linux system, to keep compatibility among several architectures, emulates such operations even on lacking ones.

address with the third parameter, it checks if the memory content matches the old value. If so, the swap is performed. Otherwise, it means that another write happened meanwhile and the CAS simply leaves the memory address untouched. Due to its nature, commonly a CAS is sited inside a loop that tries to execute the operation until success. Indeed, atomic operations turn out to be slight slower[16] than an equivalent sequence of normal operations, yet provide an excellent resource for synchronisation that in many cases overcomes the traditional techniques.

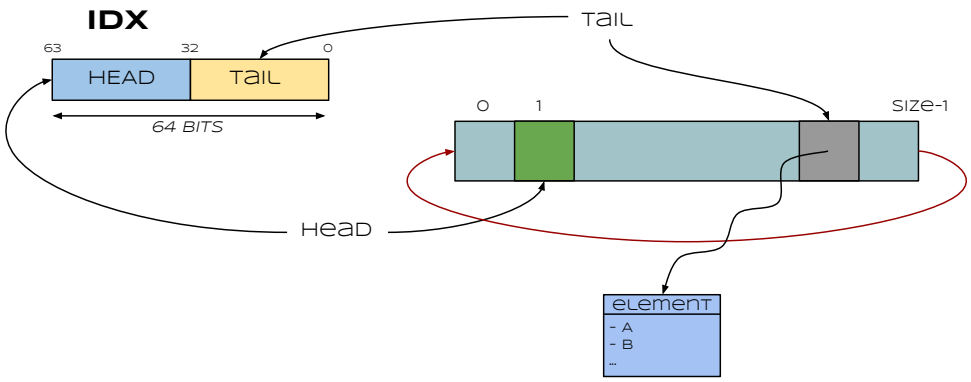

FIGURE 3.4: *General structure of the buffer*

Coming back the buffer implementation, the figure 3.4 depicts a high-level description of the data structure. The buffer implements a circular logic such as the central structure can be realised with an array and two more indices: HEAD and TAIL. The former represents the position where someone can start reading while the latter is the index reserved for a new element. The buffer follows a First-In-First-Out (FIFO) behaviour and exposes only two methods: INSERT() and REMOVE(). The former method can always be performed, even if the buffer results full. In that case, an overriding policy is applied, and the just arrived element substitutes the oldest one. The latter, instead, is for reading elements from the buffer, consequently, if it is empty, the function will fail.

The INSERT() method algorithm 3.3 is invoked during the NMI handler task, and as such it must be suitable for a fast computation. To guarantee this, we implemented a *wait-free* function that never blocks the writer while attempting to insert a new element, by designing a synchronisation logic that always gives the go-ahead to the writer.

At any moment, the buffer state can be one of the following:

- EMPTY: head and tail are equals. Only INSERT() can be performed.

[16]They directly affect the cache adding some overhead.

Algorithm 3.3 Buffer insert()

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

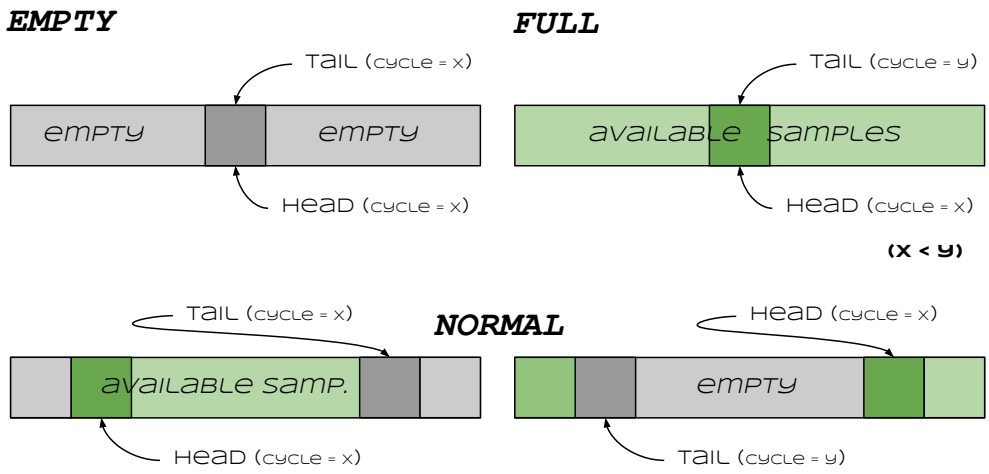

FIGURE 3.5: *All possible states of the buffer*

- FULL: head and tail are logically equal, but the tail is one cycle[17] forward. Both INSERT() and REMOVE() can be performed, but the former implies an override.

- NORMAL: head and tail are different[18], and both methods can be called.

The tail and head indices are saved into the same 64-bit variable[19] (IDX). This choice is necessary because we can save an extra read to verify certain conditions while performing the REMOVE(). The first 32-bit of the IDX represents the tail while the other half is reserved for the head. As the first operation, the INSERT() reads the IDX value and checks if the head is a cycle back the tail (FULL) (2 in 3.3). If so, it tries to perform a CAS to increase the tail (3)[20] and two cases may be possible:

- Success: Between the state audit and the increment read a sample (head untouched). The tail is advanced so that any reader is informed and potentially blocked (see remove() algorithm 3.4). Then the insert operation is started causing the old element rewrite (4). In the end, the head is incremented, and any reading action can be executed (5).

- Failure: After the index check, a reader executed the remove() function. This case brings back to the NORMAL state like the initial audit was not true.

If the first check did not succeed, then we are in the NORMAL or EMPTY state. In this situation, any reader, it does not matter how many times the remove() is invoked, eventually will stop, taking the head to the tail value (EMPTY). Thus, the writer can simply write the new element (9) then advance the tail to inform all the readers of the presence of a new sample (10). This increment must be done via a FAA operation because the IDX is shared by head and tail and possible concurrent accesses by readers and writer may occur. Even if the latter can be incremented without any trouble, the higher part of the IDX value must be consistent to concurrent reader access, so a reliable atomic operation is required. Despite the presence of atomic operations which are slightly slower than normal ones, the general execution it quite fast, and above all, it never waits.

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

[18] The logic value of head can be greater or smaller of the tail one depending on the respective cycles.

[19] On Linux x86-64 system the atomic operations can work on 64-bit variables

[20] Even though a fetch_and_add may seem more suitable, it is not because between the first read and the performed increment, atomicity is not guaranteed.

Algorithm 3.4 Buffer remove()

```
1: loop
2:     midx ← IDX
3:     while HEAD(midx) + SIZE < TAIL(midx) do
4:         midx ← IDX
5:     end while
6:     if HEAD(midx) = TAIL(midx) then
7:         return EMPTY
8:     end if
9:     elem ← READ(TAIL(midx))
10:    if CAS(IDX, midx, HEAD(midx) + 1) then
11:        return SUCCESS
12:    end if
13: end loop
```

Finally, by looking at the remove() function 3.4, we can observe the entire strategy. Recalling the reader can be blocked in case of concurrent action, it does not wait actually, but it aborts and restarts the remove() procedure. This justifies the code wrapping in a while loop. A reader must wait if the state is FULL and the tail is a step forward (`3-5` in 3.4) because the writer is overwriting the element referenced by the head. It identifies an ongoing write that is performing an overwrite, and then the read sample may not be consistent. A second step verifies the buffer is not EMPTY. If so, the method returns with EMPTY_BUFFER value (`6-8`). If the state is NORMAL or FULL, the reader can start reading the first available element (`9`). Once finished, the last condition that is verified is intended to guarantee that readers and writer did not mix their tasks (`10-12`). The CAS on the entire IDX let the reader check both the tail (writer collision) and the head (reader conflict) so that if the old value of IDX does not match the value read at the begin of the procedure, everything is lost and the loop restarted. In this model, the reader is dominated by the writer that has preemption rights. We did not investigate if this solution is suitable for a generic use case because we did not provide any analysis application able to enable some readers. Indeed, the performance depends on several factors like the sampling rate, which determines the write frequency, as well as the number of readers that increase the number of conflicts. The main target of this buffer structure implementation is providing a basic support to deal with the critical sections of the module.

CHAPTER 4

Experimental Assessment

The best way to show that a stick is crooked is not to argue about it or to spend time denouncing it, but to lay a straight stick alongside it.

— Dwight L. Moody

In this chapter we are going to show the experimental results obtained from several tests conducted to measure the performance impact of the HOP activity. All the tests have been conducted on Linux Debian 8.0 (Jessie) kernel 3.16.36 running on HP Proliant NUMA machine equipped with 4 x AMD Opteron(tm) Processor 6128 and 64 Gb of memory. This lets us benefit from the *non uniform memory access* system extending the testing phase to an increasingly common architectural organization.
In order to evaluate the performance of the proposed solution we adopted some benchmarks from the PARSEC Benchmarks Suite[BKSL08] :

- **BlackScholes** : is a Recognition, Mining, and Synthesis (RMS) application based on Black-Scholes partial differential application and used as a method for emulating general PDE programs. Its performance is bounded by the floating-point computing capability of the underlying hardware. More precisely, it simulates a set of financial calculus operation and spreads the computation among the available threads.

- **Canneal** : implements a *Simulated Annealing* (SA) algorithm using to simulate some problems in chip design. SA belongs to the class of the *local searches algorithm* which aim to find a local optimum over a big search space. This application uses sophisticated lock-free synchronization techniques and enforces its execution via a cache-aware design.

- **Fluidanimate** : is another Intel RMS application used to simulated the behaviour of a fluid with free surfaces. It has been included in the PARSEC suite because of the always increasing software tendency of exploiting physical natural simulation (e.g., computer games).

- **Swaptions** : takes advantage of the Monte Carlo simulation which represents the base for the PDE simulators. This is an RMS applications which operate in the economic context.

Each benchmark has been compiled via the PARSEC utility script such that the underlying parallelism technique[1] relies on the *pthreads library*. Additionally, every configuration has been run five times[2] so that, eventually, an average value could be computed. We employed the *time* program, natively available on Unix-based and Linux systems, to record the execution durations. In this way, we were able to obtain the *real* time along with *cpu* and *sys* time at the end of a run. Compared to the real time, the latter represents the aggregated result value of all the engaged threads. Every used benchmark has been slightly patched so that each thread, upon its creation, could register its presence to HOP by itself. The module structure lets a natural check-in phase because the profiler can keep its state (active or not) and automatically insert the thread being added to the monitoring activity. Statics are collect thread by thread to guarantee a more detailed study. PARSEC provides several datasets to be fed to the considered application. *simsmall*, *simmedium* and *simlarge* are intended for testing runs or a very light execution (also launched on a simulator), while *native*, which presents a different grade of magnitude, is used for observing the application behavior during an extended activity. We adopted the native dataset so that, besides the study in a more realistic execution context, also startup times and other secondary costs due to initial works could be considered negligible compared to the overall thread task.

Benchmark	**Parallelization**		**Data Usage**	
	Model	Granularity	Sharing	Exchange
blackscholes	data-parallel	coarse	low	low
swaptions	data-parallel	medium	low	low
fluidanimate	data-parallel	fine	low	medium
canneal	unstructured	fine	high	high

TABLE 4.1: *Key characteristics of the PARSEC benchmarks employed in this experimental phase [BKSL08].*

The bench configuration varies on the number of the used threads and the IBS sampling periods. We tested for a number o threads belonging to

[1]PARSEC provides both a serial and parallel version of a given benchmark. Several parallelism approaches are available and comprise even advanced technique like open-mp.

[2]We experienced a low deviation, namely about 2%, among the results of the deterministic-nature benchmarks.

the **[1, 2, 4, 8, 16, 32, 64, 128]** set because we wanted to study the serial execution along with the parallel one, even reaching the hardware capability saturation (introducing an extra cost for the high context switching). The frequency values range from **4096** to **32678** by an incrementing step of *4096* and are based on instructions counting rather than clock cycles counting. This choice was taken according to some experimental results illustrated in other works and those obtained in some preliminary tests that revealed the chosen frequencies to be an optimal bound. Actually, it is possible to exploit a smaller period as lower-bound such as *2048*. However, in some contexts, the high number of generated NMIs would induce a huge interruptions occurrences making the per-thread progress to go forward at very low rate. Furthermore, we observed the attitude of small sampling period may cause stalls of the CPU because of the NMI management side effects. We considered the collected samples by the 4096 sampling period as the maximum accuracy we can reach in this tests for the purpose of this investigation.

4.1 Overhead

We decided to consider the cpu and sys time combination instead that the real-time taken by a single run. This choice enabled us to study the benchmark behavior deeply while under monitoring, observing the cost of the module task too. As the cpu-time plots show, incrementing the sample period accentuates the overhead curve which always follows a main trend. As we step closer to the core number saturation point by increasing the number of employed threads, the curve slope raises. In particular, the cpu-time curve obtained during execution at low frequency rates (at least 16K instructions) keep the same ratio with the curve generated by the plain run in all the points. Intermediate cases ($12\text{K} < x < 16\text{K}$) start introducing a soft ratio increment as soon as the threads number reaches and goes over 32 (hardware limit). By setting smaller sampling periods, this phenomenon may be experienced earlier in conformity of smaller threads employment. Observing the cpu-time plot of the blackscholes test (figure 4.4), by using a 4096 sampling period, the rise starts after 8 threads. We already know that the number of NMIs remarkably increases as we lower the sampling rate. By analyzing the plain-execution curve at a microscopic level, we noticed a minimal rising curve slope so that the profiled curves potentially share the same trend. The NMIs nesting in such points exacerbates this behavior, due to the application's own nature, by several orders of magnitude. As a matter of fact, faster NMIs generation highlights with higher certainty some application weaknesses due to dominant effects such as synchronization. The bottleneck gradually tends to move toward the left part of the graph (lower number of threads) leaving its ideal location. Canneal stands out

for the profiled activity execution times that accurately follows the trend of the plain-run. The delta factor is kept constant even at high threads employment without incurring in a scalability decline. Moreover, the ratio between two periods is equal to the ratio of their time values. The overhead values are reported by the table 4.6 and represents the duration value for each sampling rate compared to the same configuration plain execution. The numbers tell that to obtain an operating cost less than 20% the period must be kept quite high (over 16K). Furthermore, the grade of accuracy is a parameter to focus on while configuring the profiler. However, this results highly depends on the kind of profiled application as it is possible to observe in some cases (e.g., canneal) overhead values which are about 15% in the worst case. Canneal is highly dominated by synchronization effects which allow us to intensify the profiling activity without linearly adding overhead. Being aware of the internal mechanisms of the looked application represents an aid element to configure the profiler activity.

4.2 Accuracy

Even though the application takes different times to complete according to the set sampling frequency, it is possible to note how the benchmark execution, over different threads configuration, provides an almost constant number of generated samples. Data-partitioning nature accounts for that outcome because even spreading the job over different threads, the amount of work to be performed is untouched. Fluidanimate differs from this property because of its internal synchronization techniques. It primarily takes advantage of lock primitives[BKSL08] which turn out to add extra work during thread task execution, influencing the sample collection.
By taking into account two sampling rates, the number of samples respectively gathered in each one follows the ratio between the two used frequency period. For instance, by looking at the curves described by 4096, 8192 and 32768 in the figure 4.8, the values comply with the gap among the rates such that 2.5 * 100M at 4096 is halved to about 1.2 * 100M at 8192, which in turn is reduced by a factor 4 at 32768, obtaining about 4 * 10M. This linear factor is critical and should be taken into account during the calibration step.

4.3 Efficiency

We investigated the efficiency of adopting a configuration as a trade-off between generated samples and produced overhead. The figure 4.12 depicts the plot derived from the number of collected samples and the cpu-times

required to complete the run. We named this *samples-per-second* (SPS) graph. The drawn curve represents the ratio between two aggregated value (all available threads outcome) and identifies the average samples contribution given by each thread per second. We already saw the number of collected samples being constant varying the parallelism degree of the application, thus what we expect is the reflected image of the cpu-time graph. As a consequence, the scalability starts progressively downgrading with lower sampling period. An interesting fact that may not easily observe in other graphs is the *goodness* attributed to each sampling rate. Goodness means the contribution a thread gives for a specific configuration. According to the data-parallel nature, we would expect the ratio of different curves to be compliant with the ratio of the considered sampling period values. It is not. The cost due to secondary effect and other background jobs exponentially increase for lower periods. This side-effect makes the SPS value generated by certain periods not to produce a benefit with respect to slower frequencies as the parallelism rises. Consequently, this property may be useful for a runtime tuning of the monitoring activity so that the wanted SPS value can always be satisfied. Canneal is the only one that shows a nearly optimal behavior that perfectly scales according to threads and period variation. As matter of fact, the graph depicts almost constant SPS curves that keep that ratio untouched.

4.4 Bottleneck

Threads	**Full**	**No Sample**	**Void**
1	70%	41%	40%
2	89%	59%	55%
4	89%	62%	61%
8	98%	70%	69%
16	204%	200%	196%
32	461%	446%	437%
64	398%	367%	361%
128	348%	307%	306%

TABLE 4.2: *CPU times overhead incurred while executing blackscholes subject to a 4096 freq. rate profiling analysis.* ***Full***, ***No Sample*** *and* ***Void*** *are different NMI handler routine implementations. This evaluation has been conducted on the* simlarge *dataset.*

Besides the studies conducted above, we also wanted to analyse the impact of our implementation choices regarding the IBS support. In particular, we evaluate the repercussion of the NMI management during sampling activity. As largely described in the previous chapter, the NMI generation represents the only way to notify the availability of a new IBS sample. We reduced the job performed by the NMI handler routine by removing secondary operations from those used in the final logic. Notably, a test has been configured limiting the handler activity for just managing the IBS interrupt(lines `1-6` in the algorithm 3.2) and returning the control without taking any further operation. We label this handler function implementation *Void*. A second test performed the full logic function without managing the sample collection (it does not executes lines `19-21`), thus saving time in buffer writing and data reading. This implementation is called *No Sample*. The table 4.2 extracts some results from the cpu-times overhead comparison of these two tests and the *full* execution. Although the time values reduce with the logic simplification (the most expensive section is the sample write), the overhead is definitively still high. It is possible to conclude that the significant number of NMI occurrences is the primary source of slow down (related to secondary events such as cache pollution and pipeline flushing during context switches) that affects whatever IBS-based solution.

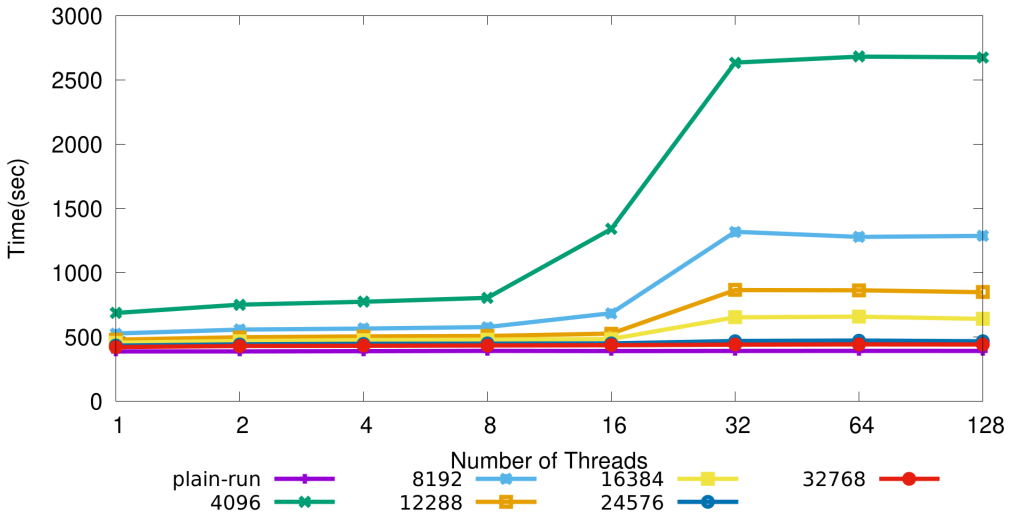

FIGURE 4.1: *Blackscholes: cpu-time execution*

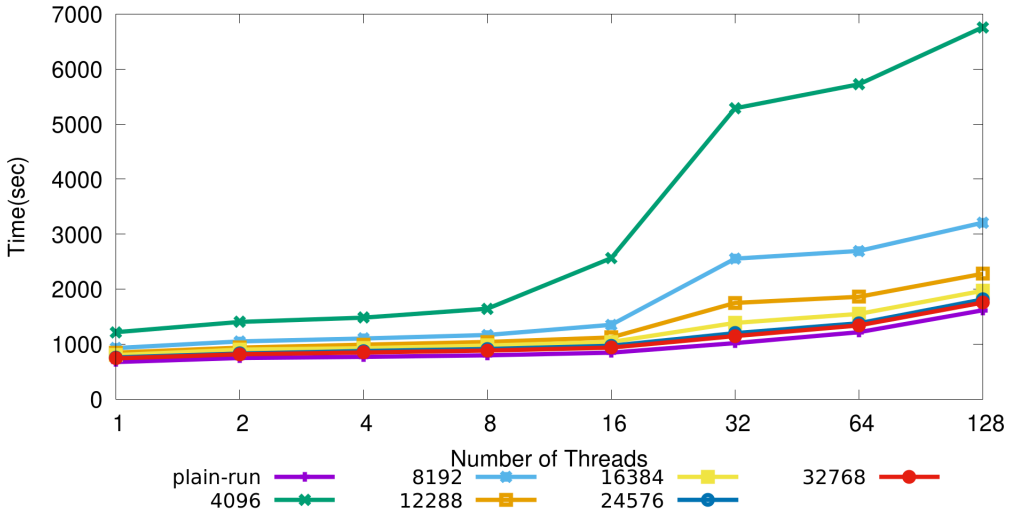

FIGURE 4.2: *Fluidanimate: cpu-time execution*

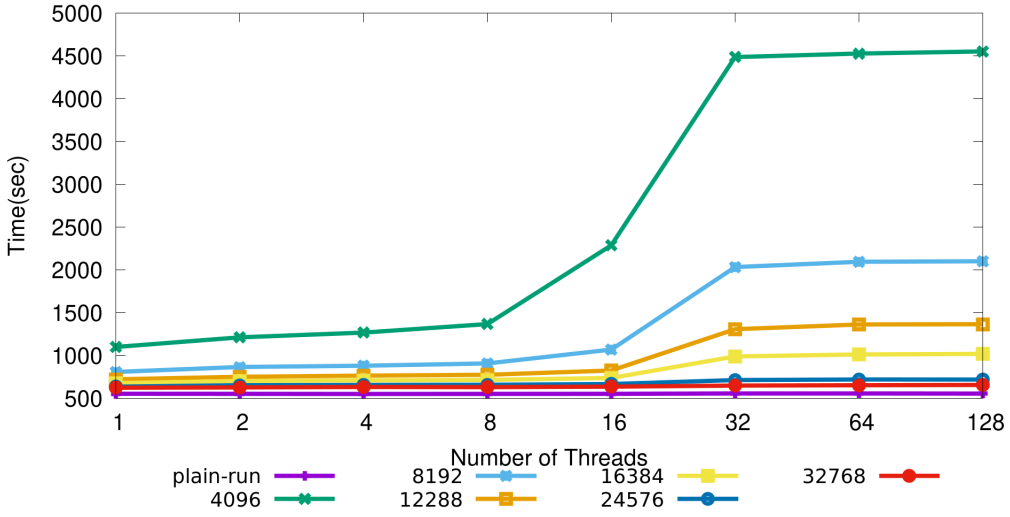

FIGURE 4.3: *Swaptions: cpu-time execution*

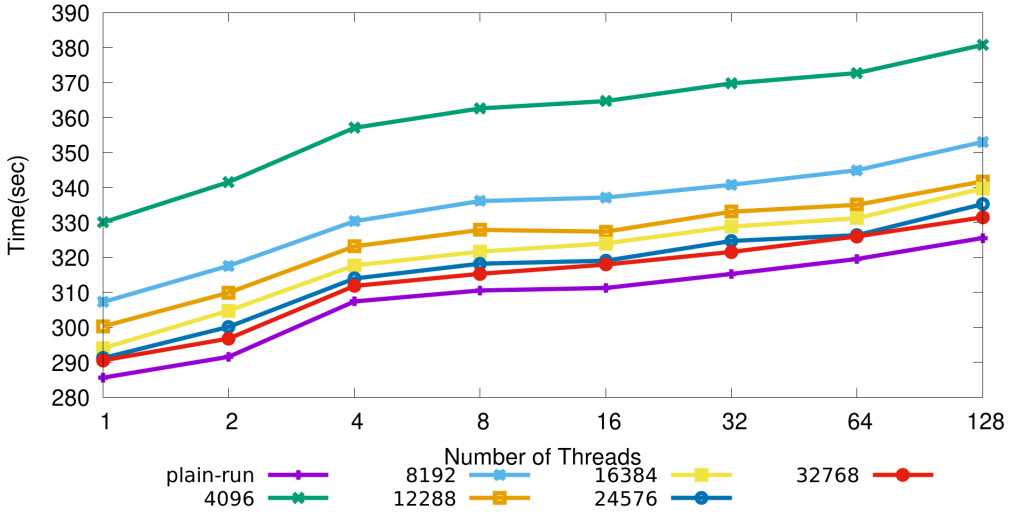

FIGURE 4.4: *Canneal: cpu-time execution*

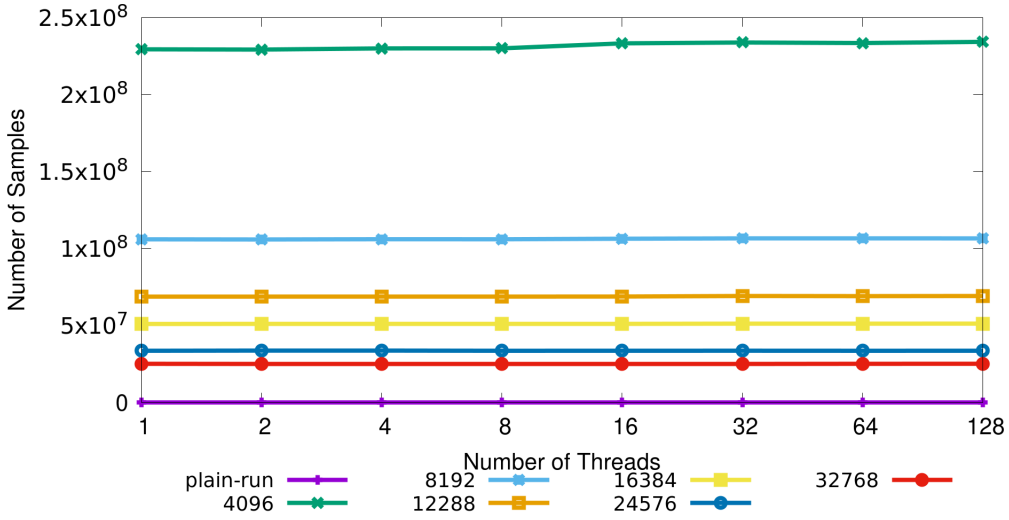

FIGURE 4.5: *Blackscholes: number of generated samples*

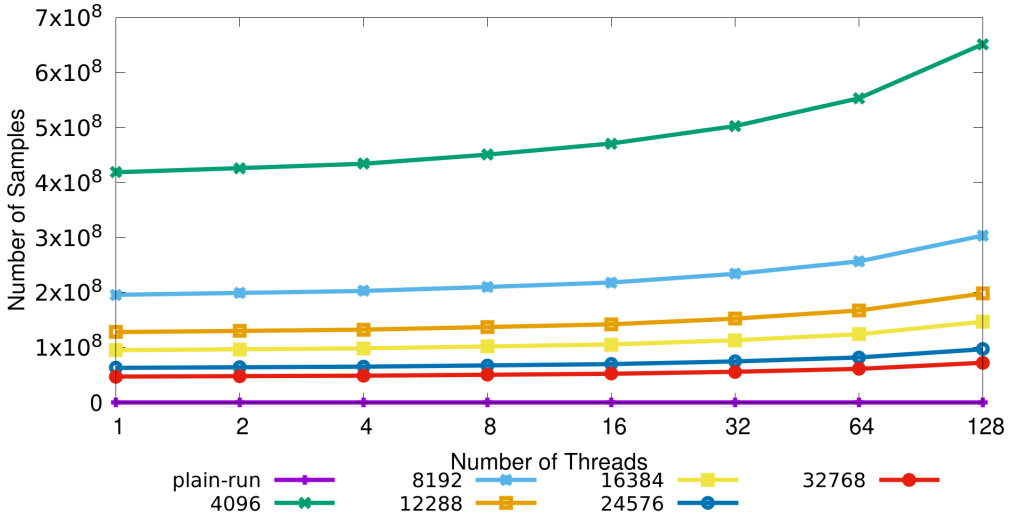

FIGURE 4.6: *Fluidanimate: number of generated samples*

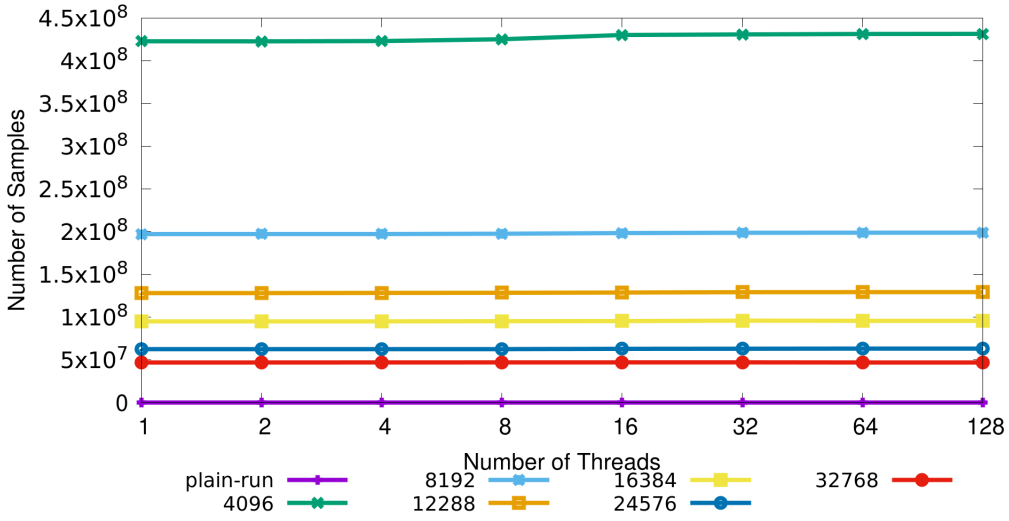

FIGURE 4.7: *Swaptions: number of generated samples*

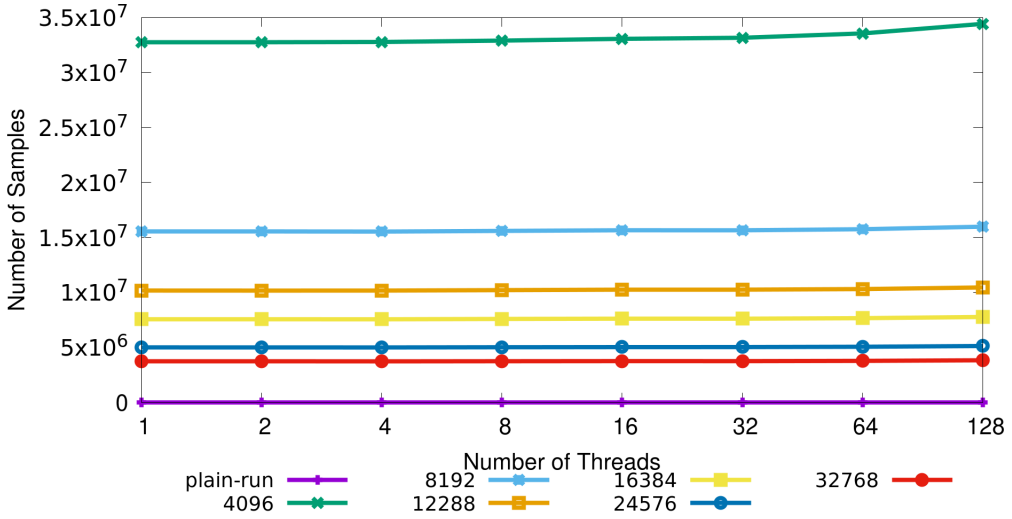

FIGURE 4.8: *Canneal: number of generated samples*

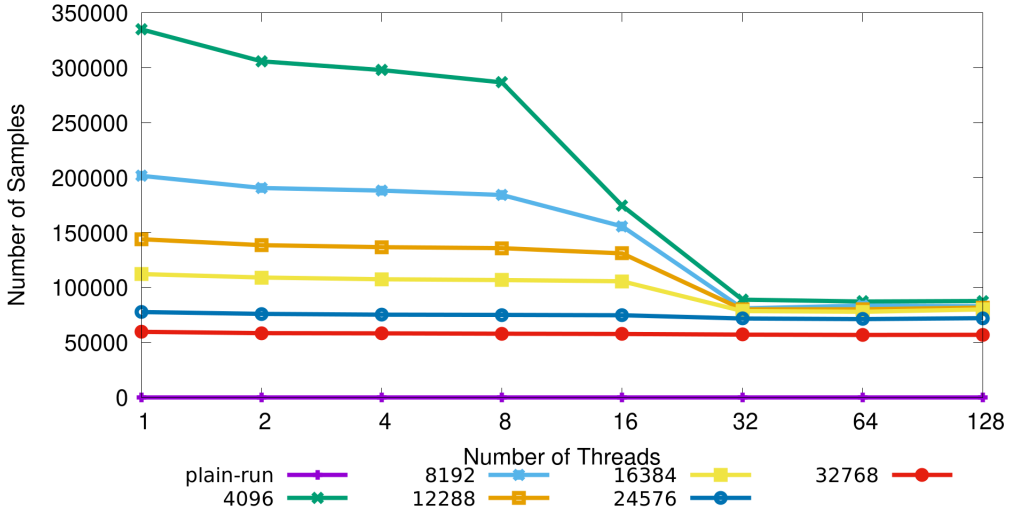

FIGURE 4.9: *Blackscholes: per-second generated samples over cpu-time execution*

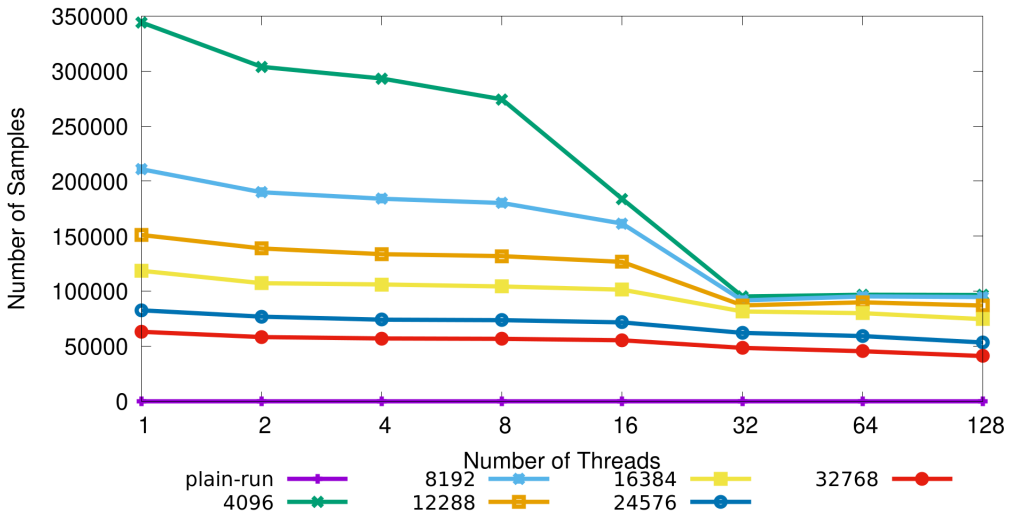

FIGURE 4.10: *Fluidanimate: per-second generated samples over cpu-time execution*

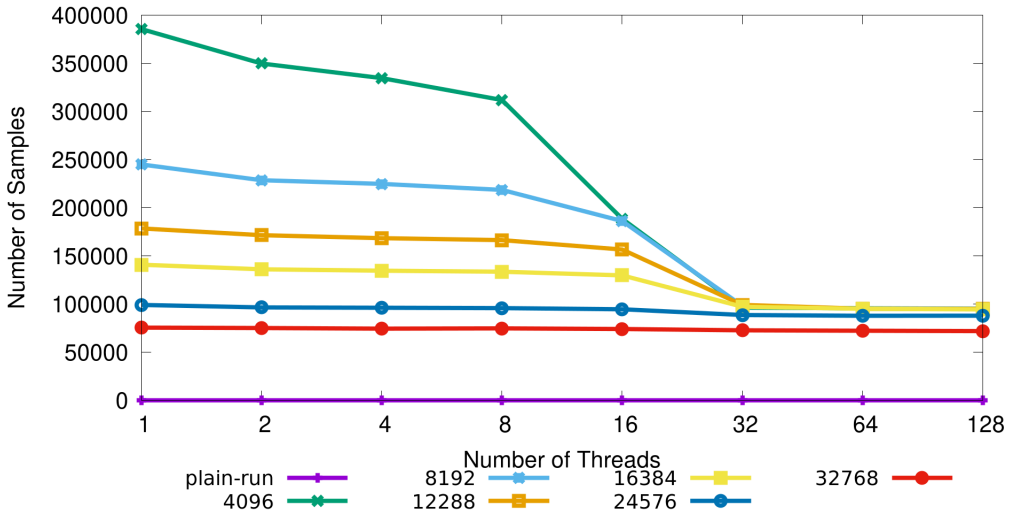

FIGURE 4.11: *Swaptions: per-second generated samples over cpu-time execution*

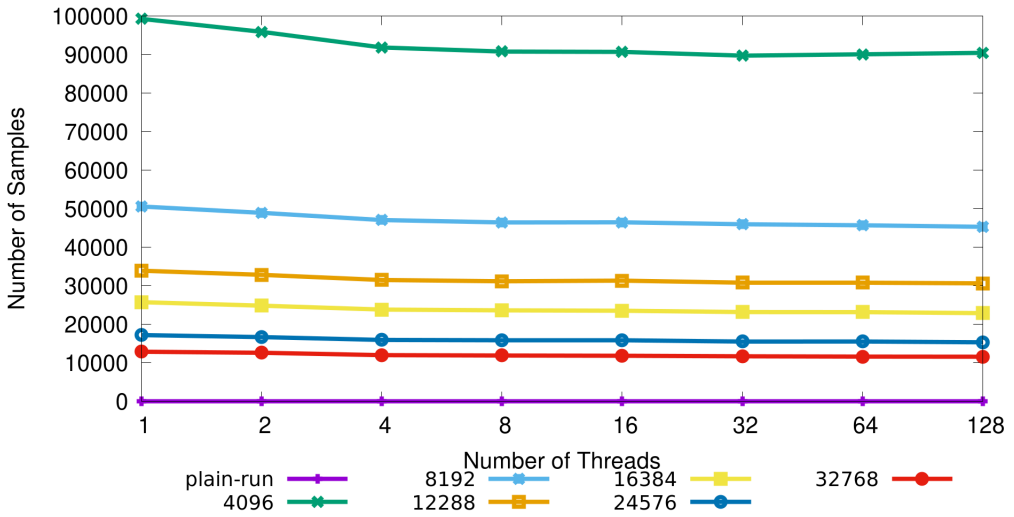

FIGURE 4.12: *Cannel: per-second generated samples over cpu-time execution*

Period	**Threads Number**							
	1	2	4	8	16	32	64	128
4096	77.08	93.84	99.30	106.0	244.8	576.97	587.91	587.05
8192	35.79	43.60	45.39	47.63	75.99	238.46	227.75	230.00
12288	23.34	28.20	29.69	29.79	35.13	121.84	121.05	117.34
16384	17.24	20.70	22.24	22.42	24.30	67.31	68.37	64.05
24576	11.42	13.96	14.83	14.47	15.44	19.97	20.63	19.31
32768	8.37	10.66	10.84	10.82	11.72	12.63	12.96	12.88

TABLE 4.3: *Blackscholes: overhead percentage according to a plain cpu-time execution.*

Period	**Threads Number**							
	1	2	4	8	16	32	64	128
4096	80.77	87.90	92.93	106.7	203.76	419.76	370.84	318.7
8192	37.86	40.39	43.49	46.57	60.00	150.98	121.54	98.64
12288	25.61	25.30	28.87	30.56	32.77	71.85	52.96	41.12
16384	18.80	20.38	20.45	22.47	22.99	36.11	27.34	21.97
24576	12.49	10.90	13.86	14.47	14.86	17.77	13.41	12.21
32768	10.00	9.07	10.33	10.97	10.81	12.54	10.14	8.66

TABLE 4.4: *Fluidanimate: overhead percentage according to a plain cpu-time execution.*

Period	**Threads Number**							
	1	2	4	8	16	32	64	128
4096	99.50	120.2	131.1	148.8	316.4	714.31	722.03	724.89
8192	46.36	57.19	60.31	64.92	94.26	268.63	280.01	280.33
12288	30.70	36.30	39.16	40.74	49.77	136.84	146.90	146.91
16384	22.71	27.24	29.01	29.87	33.82	79.04	83.24	84.09
24576	15.07	18.33	19.16	19.44	20.94	28.86	30.11	29.71
32768	12.80	13.65	14.94	14.34	15.34	17.15	17.96	18.34

TABLE 4.5: *Swaptions: overhead percentage according to a plain cpu-time execution.*

Period	**Threads Number**							
	1	2	4	8	16	32	64	128
4096	15.55	17.15	16.17	16.77	17.18	17.29	16.65	16.97
8192	7.58	8.93	7.47	8.24	8.31	8.09	7.95	8.44
12288	5.13	6.30	5.13	5.58	5.18	5.65	4.85	4.98
16384	2.95	4.53	3.35	3.58	4.10	4.30	3.65	4.35
24576	1.97	2.94	2.13	2.47	2.51	3.00	2.13	2.98
32768	1.73	1.79	1.45	1.54	2.15	2.00	2.02	1.82

TABLE 4.6: *Canneal: overhead percentage according to a plain cpu-time execution.*

CHAPTER 5

Conclusions and Future Work

If you want to go somewhere, `goto` *is the best way to get there.*
— Ken Thompson

In this thesis we presented a new solution for on-line profiling of applications based on the AMD Instruction-based Sampling feature. The main idea of this proposal is using the hardware support provided by most of modern processors to obtain information about the application execution. This technique is almost transparent to the analysed software because the only required operation is the process registration for the monitoring activity. However, this may be directly performed by the user or another process. The performance monitor units provide a high precision measurement of hardware events—for instance cache miss/hit events o number of retired instructions—that software-based supports might only estimate. To obtain the best efficiency and the maximum control of underlying hardware, the communication with IBS registers is direct. Moreover, the module configures both the PMU and the operating system in order to handle the interruption upon sample generation (IBS interrupts). HOP works on thread context discriminating the activity of the execution flow of interest during the monitoring. To achieve such a capability, we directly operate on the context switch system function through an external module, so that PMU activity can be activates or deactivated when needed. Our solution highlights a delicate context when working with PMU. As a matter of fact, the samples collection occurs in the interrupt context, which requires specific precautions in terms of execution complexity. In such a context the performed routine should be as light as possible and furthermore, it must never block. This means that synchronization mechanisms have to be sophisticated and not rely on blocking technique like spinlocks. For each registered thread the application builds a ring-buffer that keeps the generated samples, whose access is managed through non-blocking atomic operations. Several tests have been conducted to evaluate the goodness of our module and the IBS subsystem. The generated overhead is highly dependent on the configured sampling frequency, the number of profiled threads in the system and

the application workload. In particular the highest application parallelism degree tends to move backwards on a smaller value than the number of available cores when increasing the frequency rate. Generally, this behaviour is due to the *cpu-bounded* nature of the application. *Memory-bounded* application tests present an overhead of about 15% in the worst cases. Extra tests have been carried out to study the IBS support implementation. We analysed the application behaviour change — in terms of slowdown — when varying the tasks performed by the IBS NMI handler. By adopting a minimal handling job, that is the mere IBS interrupt catching without executing other logic, the overhead curve follows the same trend of the full implementation used in the module. Indeed, this clearly identifies an intrinsic deficit of IBS which will be reported on all the solutions based on such an approach. We already planned several improvement for the future version of HOP:

- *more architectures*: we are going to extend the set of architectures which can be supported. In particular, we would like exploring the Intel PEBS. It works on the events domain and is enhanced by an advanced mechanism that allows samples buffering at firmware level. Saving an interrupt generation for each sample occurrence may be extremely useful and certainly lowers the incurred slow-down.

- *event set*: currently HOP is a general-purpose profiler. This implies that the type of the observed events cannot be customized, thus it not possible to specialise its activity. We would like to provide an interface for supporting specific events such as memory-related ones.

- *autonomous rate*: although HOP already provides a facility for at runtime tuning of the sampling rate, we would like introducing an automatic management of the sampling rate. This may reduce the probability of throttling events and provide a more advanced form of sampling. In some cases, it may be preferable to intensify the observation during a specific execution period and keep it as low as possible in other situations. This logic can be paired with an ideal overhead value that represent the maximum slowdown admitted for that profiling session.

- *specialization*: besides the general-purpose nature for building sophisticated analysis system, we think that directly specializing HOP for a specific context would turn out to be extremely useful. We are going to instantiate a particular module structure in order to take advantage of the profiling activity for optimizing NUMA systems. The low-profile monitoring activity would identify sub-optimal executions such that optimization actions like page or thread migration may be undertaken.

Bibliography

[AH17] Soramichi Akiyama and Takahiro Hirofuchi. 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 2017*, ROSS '17, pages 3:1–3:8, New York, NY, USA, 2017. ACM.

[BDG+00] Shirley Browne, Jack J. Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *IJHPCA*, 14(3):189–204, 2000.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, pages 72–81, 2008.

[BN14] G. Bitzes and A. Nowak. The overhead of profiling using pmu hardware counters. In *CERN openlab report*, 2014.

[Coh04] William E. Cohen. Tuning programs with oprofile. In *Wide Open Magazine 1*, pages 53–62, 2004.

[Cor16] IBM Corporation. Getting started with oprofile. 2016. Available at: https://www.ibm.com/support - Accessed: 2017-12-29.

[Cor17] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2017.

[CVH+10] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 42–52, 2010.

[Dev10a] Advanced Micro Devices. *AMD64 Technology - Lightweight Profiling Specification*. 2010.

[Dev10b] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. 2010.

[Dev11] Advanced Micro Devices. *Software Optimization Guide for AMD Family 10h and 12h Processors*. 2011.

[Dev17] Advanced Micro Devices. *AMD64 Technology - AMD64 Architecture Programmer's Manual Volume 2: System Programming*. 2017.

[DFF+13] Mohammad Dashti, Alexandra Fedorova, Justin R. Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 381–394, 2013.

[DHW+97] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Z. Chrysos. *ProfileMe*: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 30, Research Triangle Park, North Carolina, USA, December 1-3, 1997*, pages 292–302, 1997.

[Era08] Stéphane Eranian. Perfmon2 : a standard performance monitoring interface for linux. 2008. Available at: http://perfmon2.sourceforge.net/perfmon2-20080124.pdf - Accessed: 2017-12-29.

[Gho16] Amir Reza Ghods. *A Study of Linux Perf and Slab Allocation Sub-Systems*. 2016.

[Gre17] Brendan Gregg. Linux performance, 12 2017. Available at: http://www.brendangregg.com/linuxperf.html - Accessed: 2017-12-30.

[JJN08] S Jarp, R Jurga, and A Nowak. Perfmon2: a leap forward in performance monitoring. *Journal of Physics: Conference Series*, 119(4):042017, 2008.

[KTC01] W Korn, Patricia Teller, and Gilbert Castillo. Just how accurate are performance counters? pages 303 – 310, 05 2001.

[Lab] Hewlett-Packard Laboratories. The pfmon tool and the libpfm library. Available at: http://perfmon2.sourceforge.net - Accessed: 2017-12-29.

[LLQ12] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. Memprof: A memory profiler for NUMA multicore systems. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 53–64, 2012.

[LMW15] Ivonne López, Shirley Moore, and Vincent M. Weaver. A prototype sampling interface for PAPI. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure, St. Louis, MO, USA, July 26 - 30, 2015*, pages 27:1–27:4, 2015.

[LTCS10] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: efficient static binary instrumentation for linux. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, pages 175–183, 2010.

[Man17] Linux Programmer's Manual. Perf_event_open(2), 09 2017. Available at: http://man7.org/linux/man-pages/man2/perf_event_open.2.html - Accessed: 2017-12-30.

[Moo02] Shirley Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *Computational Science - ICCS 2002, International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part II*, pages 904–912, 2002.

[MSHN17] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E. Nagel. Detecting memory-boundedness with hardware performance counters. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 27–38, 2017.

[MV10] Collin McCurdy and Jeffrey S. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, pages 87–96, 2010.

[OPr17] Oprofile, 2017. Available at: http://oprofile.sourceforge.net - Accessed: 2017-12-29.

[Pel13] Alessandro Pellegrini. Hijacker: Efficient static software instrumentation with applications in high performance computing: Poster paper. In *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*, pages 650–655, 2013.

[Pet11] M. Pettersson. The perfctr interface, 2011. Available at: http://user.it.uu.se/ mikpe/linux/perfctr - Accessed: 2018-01-01.

[PQ17] Alessandro Pellegrini and Francesco Quaglia. A fine-grain time-sharing time warp system. *ACM Trans. Model. Comput. Simul.*, 27(2):10:1–10:25, 2017.

[Ros12] Steven Rostedt. The x86 nmi iret problem, 2012. Available at: www.lwn.net/Articles/484932 - Accessed: 2017-09-30.

[SA00] Harsh Sharangpani and Ken Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, 2000.

[SMM16] Manuel Selva, Lionel Morel, and Kevin Marquet. numap: A portable library for low-level memory profiling. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016, Agios Konstantinos, Samos Island, Greece, July 17-21, 2016*, pages 55–62, 2016.

[SS10] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 33–46, 2010.

[TWF17] Roman Dementiev Thomas Willhalm and Patrick Fay. Intel performance counter monitor - a better way to measure cpu utilization, 2017. Available at: https://software.intel.com/en-us/articles/intel-performance-counter-monitor - Accessed: 2018-01-01.

[ZJH09] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pages 23–32, 2009.