KerMon: Framework for in-kernel performance and energy monitoring

Diogo Ricardo Cardoso Antão

Thesis to obtain the Master of Science Degree in
Engenharia Electrotécnica e de Computadores

Supervisors: Doutor Pedro Filipe Zeferino Tomás
Doutor Aleksandar Ilic

Examination Committee

Chairperson: Doutor Nuno Cavaco Gomes Horta
Supervisor: Doutor Pedro Filipe Zeferino Tomás
Members of the Committee: Doutor João Nuno de Oliveira e Silva

November of 2014
"Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time." Thomas Edison
Acknowledgments

Firstly, I need to thank Prof. Leonel Sousa for being a great teacher and mentor. Especially for allowing me this opportunity to do the work presented in this thesis and for never giving up on me. Of course, this thesis would not have been possible without the support and coordination of Pedro Tomás. I am heartily thankful to Aleksandar Ilić, whose encouragement, guidance, support and especially patience from the initial to the final stage enabled me to develop an understanding of the subject. I also must thanks Frederico Pratas and Luís Taniça for their contribution on the subject of this thesis.

I owe my deepest gratitude to my colleagues at SIPS for the support and help. I was to name a few people, but that few is actually a lot of people. So to be fair: If we know each other, you have my thanks. I am grateful to my parents for providing the necessary education in order to reach this step of my life. I want to thank my friends for the encouragement, namely Ana, Jelena (both of them), Lisandra, Hirak, Svetislav, that have to hear me and Aleksandar discuss my thesis.

The work presented herein was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects Threads (ref. PTDC/EEA-ELC/117329/2010) and P2HCS (ref. PTDC/EEI-ELC/3152/2012).

Finally, to all of those whom I do not mention here but that directly or indirectly have contributed and helped me to achieve my goals,
Thanks!
Abstract

Accurate on-the-fly characterization of application behavior requires assessing a set of execution related parameters at runtime, including performance, power and energy consumption. These parameters can be obtained by relying on hardware measurement facilities built-in modern multi-core architectures, such as performance and energy counters. However, current operating systems (OSs) do not provide the means to directly obtain these characterization data. Thus, the user needs to rely on complex custom-built libraries with limited capabilities, which might introduce significant execution and measurement overheads. Moreover, none of these libraries allow its usage by kernel facilities such as the task scheduler. In this work, we propose a low overhead monitoring framework that relies on kernel-space facilities and user-space tools to capture the run-time behavior of a wide range of applications. By relying on the task scheduler to directly capture the characterization information, the proposed KerMon framework allows improving the quality of application monitoring, by reducing the overall overheads and minimizes the impact on the processor (e.g., cache) state. This information can be used to analyze, characterize and identify bottlenecks of a system/application, such as with the Cache-aware Roofline Model (CARM), and can pave the way for new kernel scheduling strategies.

Keywords

Performance and Power Monitoring, Multi-core Architectures, Task Scheduler, Kernel, Application Characterization, Cache-aware Roofline Model
Resumo

Uma caracterização fiável do comportamento de aplicações em tempo real requer a avaliação de um conjunto de parâmetros dinâmicos relacionados com a execução, que inclui performance, potência e consumo de energia. Estes parâmetros podem ser obtidos através de mecanismos de medição embutidos no hardware das arquiteturas multi-core modernas, tais como os contadores de performance e energia. Contudo, os sistemas operativos (SOs) atuais não disponibilizam os meios para obter diretamente estes dados de caracterização. Assim sendo, o utilizador necessita de utilizar complexas bibliotecas customizadas com capacidades limitadas, que podem introduzir overheads na execução e medição. Para além disto, nenhuma destas bibliotecas permitem o seu uso pelos mecanismos do núcleo como o escalonador de tarefas. Neste trabalho, é proposto um framework de monitorização, com um overhead diminuto, que usa os mecanismos do núcleo e ferramentas user-space para obter o comportamento em execução de uma larga variedade de aplicações. Ao usar o escalonador de tarefas para obter diretamente a informação de caracterização, o proposto KerMon framework permite melhorar a qualidade da monitorização de aplicações, através da redução dos overheads gerais e minimizar o impacto no estado do processador (e.g., cache). Esta informação pode ser usada para analisar, caracterizar e identificar bottlenecks de um sistema/aplicação, como acontece com o Cache-aware Roofline Model (CARM), e pode abrir caminho para novas estratégias de escalonamento no núcleo.

Palavras Chave

Monitorização de Performance e Energia, Arquiteturas Multi-core, Escalonador de tarefas, Núcleo, Caracterização de Aplicações, Cache-aware Roofline Model
# Contents

**1 Introduction**

1.1 Motivation .................................................. 2
1.2 Objectives ...................................................... 3
1.3 Dissertation outline ........................................... 5

**2 State of the art**

2.1 Performance and Energy Counters ............................ 9
   2.1.1 Low-Level Libraries ................................... 10
   2.1.2 High-Level Libraries and Tools .......................... 11
2.2 Task Scheduler ............................................... 12
2.3 Linux Task Scheduler .......................................... 15
   2.3.1 The Deadline scheduling algorithm .................... 19
   2.3.2 Real-Time Scheduling Class .............................. 20
   2.3.3 Completely Fair Scheduler (CFS) Scheduling algorithm .. 21
2.4 Cache-Aware Roofline Model ................................ 24
2.5 Related work .................................................. 25
2.6 Summary ....................................................... 26

**3 The KerMon framework** ........................................ 27

3.1 Overview ..................................................... 28
3.2 Considerations of programming on the task scheduler context .......................... 30
3.3 Low Level access to Hardware Counters ........................ 31
   3.3.1 Raw Events ............................................ 31
   3.3.2 Event Mux ............................................... 34
3.4 Event Logger .................................................... 39
   3.4.1 Sliding Window ......................................... 40
   3.4.2 Ring Buffer ............................................. 43
3.5 KerMon scheduling classes: Scheduler++ and CFS++ .................. 46
3.6 User-Space tools ............................................. 51
   3.6.1 libSched++ ............................................. 52
Contents

3.6.2 kernelCount ................................................................. 53
3.7 Summary ............................................................................ 54

4 Experimental Results ............................................................... 57
  4.1 Experimental Environment .................................................... 58
  4.2 Overhead Analysis ............................................................... 60
  4.3 Performance Monitoring ....................................................... 63
  4.4 Energy Monitoring .............................................................. 71
  4.5 Summary ............................................................................ 72

5 Conclusions ................................................................................ 75
  5.1 Future work ........................................................................ 77
List of Figures

2.1 Modern Intel quad-core Central Processing Unit (CPU) with Hyper-threading. . . . 8
2.2 Scheduler timeline of task $T$ on a single Logical Core (LC). The task $T$ is created
on A and exits on I. The gray area represents the time when the task was running,
while the dotted lines represent scheduler ticks. ............................... 14
2.3 Example of Figure 2.2 including the invocation of task $T$ scheduling handlers. To
avoid burden the figure, it is not shown that every scheduler tick, represented the
dotted lines, invokes the task_tick handler. .............................. 16
2.4 Scheduling class iteration to select a task to be executed. .......................... 18
2.5 A task's state through the its lifecycle. ...................................... 18
2.6 Time-slice calculation procedure when a task is inserted into a CFS runqueue. 21
2.7 Task Lifecycle on CFS. .................................................... 23
2.8 Simplified example of a runqueue implemented by a Red-black tree. .......... 23
2.9 Performance Cache-aware Roofline Model of an Intel 3770K CPU. .......... 25
3.1 This Figure depicts KerMon components, represented by numbered round-corner
boxes, in an Intel/Linux computer platform. .................................. 29
3.2 Illustration of a double kernel panic. A fatal error (page fault) occurred inside the
task scheduler context. The page_fault handler kills the task and invokes the
scheduler. Since it was already running in the scheduler context, the scheduler
detects this situation and provokes a second kernel panic. ................ 32
3.3 Top part of Figure 3.1 where it shows that the Raw Events component connects
KerMon to the hardware. ..................................................... 33
3.4 Flowchart representing a simple usage model of Raw Events to set and read a
performance counter. ....................................................... 33
3.5 Top part of Figure 3.1 where it shows the relation among the KerMon components
Event Mux and Raw Events, with the underlying hardware. ................ 34
3.6 Example of event time-multiplexing of $n$ event sets. On situation 1, the task exe-
cution is suspended, thus, Period 2 is only completed the next time that the task
resumes execution. ............................................................. 35
List of Figures

3.7 Flowchart representing the usage model of the Event Mux component. 37
3.8 Simplified Unified Modeling Language (UML) diagram of the multiplexing data structure. 38
3.9 Section of Figure 3.1 where it shows the invocation of Event Mux component by the Event Logger. 39
3.10 Timeline showing a sliding window with SCHEDULERPP_SLIDING_WINDOW = 2 slots. Initially, the Sliding Window is empty (A), then the samples 0 and 1 are stored (B) (C), the window becomes full. Thus additional samples will slide the window, first to (D) and then to (E). 40
3.11 Implementation of the Sliding Window with SCHEDULERPP_SLIDING_WINDOW = 8 slots. The cursor points to the slot to be written and it loops when it reaches the end of the buffer. 41
3.12 Detail of Figure 3.17: Task execution flow on Scheduler++. The gray rectangles depict task scheduler modifications, in order to invoke Sliding Window functionality. 42
3.13 Example of a ring buffer with 8 slots. It has 2 cursors: The Head points to the slot to be written while the Tail points to first non-consumed data. When anyone points to the slot 7 and it is incremented, it will loop back to point to the slot 0. 44
3.14 Usage model of Ring Buffer functions. 45
3.15 General algorithm to pick a task to be executed. The dashed elements are implemented by KerMon. 47
3.16 Section of Figure 3.1 where it shows the invocation of Event Logger functionality by the Scheduler++. 48
3.17 Task Lifecycle on Scheduler++. This figure includes the gray boxes that depicts invocation of Sliding Window functions. 48
3.18 Example of a timeline of two tasks sharing a LC where task A executes on Scheduler++ while B executes on Fair scheduling class. In this example, the sampling period is 50 ms and the event multiplexing switching is disabled. 49
3.19 Usage model for the KerMon scheduling classes. 50
3.20 Section of Figure 3.1 where it shows KerMon user-space tools: libSched++ and kernelCount. 52
3.21 Composed preset event example, where the preset event is composed by operations among multiple raw events. The operations are performed in the left-to-right order. 53
3.22 kernelCount flowchart. 54
4.1 Total execution time difference (%) between Scheduler++ and CFS++. 61
List of Figures

4.2 Performance overheads introduced by the most invoked KerMon functions, with compiler optimizations (O2) and without (O0). .................................................. 62

4.3 Cache-aware Roofline Model (CARM) plot of the floating-point SPEC CPU2006 benchmarks. The application color characterization was made according to the dominant instruction type (DBL, Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX)). .................................................. 65

4.4 Evaluation of SPEC CPU2006 benchmarks, namely calculix and tonto, using CARM plots. .................................................. 66

4.5 Detail of Figure 4.4(b) CARM plot for the tonto benchmark. .................................................. 66

4.6 Evaluation of SPEC CPU2006 benchmarks, namely calculix and tonto, using the CARM over time. .................................................. 67

4.7 Evaluation of scalar-dominant SPEC CPU2006 benchmarks, namely gamess_3 and namd, using the CARM .................................................. 67

4.8 Multiplexing switch intervals comparison using the zeusmp and sphinx3 benchmarks. .................................................. 68

4.9 Comparison of SPEC CPU2006 benchmarks running alone or with other applications, on any LC .................................................. 69

4.10 Distribution of zeusmp and tonto among LCs, when they are executing simultaneously. .................................................. 70

4.11 Comparison of tonto benchmark running alone or with other applications, on any LC using the CARM over time .................................................. 70

4.12 Average package and PP0 power of the floating-point SPEC CPU2006 benchmarks. .................................................. 71

4.13 Package power consumption over time. The application color characterization was made according to the dominant instruction type (DBL, SSE or AVX). .................................................. 72

4.14 Comparison of tonto benchmark running alone or with other applications, on any LC using package power consumption over time. .................................................. 72
List of Tables

2.1 Scheduling policies and classes relationship [1] ........................................ 17
3.1 Data fields for the sample structure .......................................................... 39
3.2 Main compile-time definitions present under the Kconfig menu entry Scheduler++ 51
4.1 Benchmarking environment configuration ................................................. 58
4.2 Main compile-time definitions present under the Kconfig menu entry Scheduler++ 59
4.3 Configuration of the required events for GARM analysis ......................... 64
List of Acronyms

OS  Operating System

POSIX  Portable Operating System Interface

PID  Process IDentifier

API  Application Programming Interface

UML  Unified Modeling Language

CFS  Completely Fair Scheduler

FIFO  'First In - First Out'

RT  Real-Time

EDF  Earliest Deadline First

CBS  Constant Bandwidth Server

DRAM  Dynamic Random Access Memory

IO  Input-Output

RAPL  Running Average Power Limit

CPU  Central Processing Unit

GPU  Graphical Processor Unit

SMP  Symmetric Multi-Processing

SMT  Simultaneous Multi-Threading

ALU  Arithmetic logic unit

FPU  Floating-point unit

LC  Logical Core

HC  Hardware Counter
List of Acronyms

MSR  Model-Specific Register
TSC  Time-Stamp Counter
IMC  Integrated Memory Controller
QPI  QuickPath Interconnect
LLC  Last Level Cache
NMI  Non Maskable Interrupt
SSE  Streaming SIMD Extensions
AVX  Advanced Vector Extensions
PMU  Power Management Unit
PAPI  Performance API
FLOP  FLoating point Operation
FP  Floating Point
OI  Operational Intensity
ORM  Original Roofline Model
CARM  Cache-aware Roofline Model
Introduction

Contents

1.1 Motivation ................................................. 2
1.2 Objectives ................................................. 3
1.3 Dissertation outline ....................................... 5
1. Introduction

1.1 Motivation

To satisfy with the performance requirements of modern applications, modern commodity computer systems are becoming ever increasing complex machines. Initially, the performance of CPUs was increased by relying on the exponential growth of CPU clock frequency that, usually, translates to the ever-increasing number of instructions executed per second. However, this lead to an exponential and unsustainable growth in power consumption. Accordingly, the clock frequency has reached a ceiling with the current manufacturing techniques, thus the clock frequency growth is stalled. Manufacturers had to find a solution to increase performance without increasing the clock frequency. They concluded that the solution is to invest on parallelism. Hence, nowadays, the CPU performance growth is mainly due to the increasing number of CPU cores working in parallel, where each core contains the essential execution units and caches necessary for executing an instruction stream, and the use of specialized vector instructions for exploiting fine-grained parallelism at the level of each running thread. Therefore, modern CPUs are capable of Symmetric Multi-Processing (SMP) by having multiple cores. Moreover, a single core by itself is capable of SMP due to the Hyper-threading technology that allows a core to simultaneously execute 2 instruction streams.

Multi-core CPUs have a more complex architecture when compared to single-core processors, especially in the memory subsystem. For instance, the Intel Haswell CPU architecture defines a memory hierarchy composed of 4 levels: L1, L2, L3 caches and the Dynamic Random Access Memory (DRAM). The size, latency, throughput, access and sharing properties are different among them, where the latter levels are larger and slower than the former ones. For instance, the L1 cache has the smallest size, smallest latency and the better throughput than the other levels. In the other hand, the DRAM has the largest size, at the cost of also having the biggest latency and the worse throughput. This CPU architecture becomes even more complex, when considering that each core has its own private L1 and L2 cache, while the L3 cache and DRAM are shared among the cores.

Due to the above reasons, analyzing the performance of a multiprocessing system is a very complex and arduous assignment, especially since the inner-working is a trade secret. Modern CPUs provide Hardware Counters (HCs), a facility that allows to count certain hardware events, such as the number of instructions executed, cache misses, branch misses and floating-point operations. This facility can, and should be used in order to characterize and analyze the overall performance of applications and computer systems.

The characterization of applications is useful for improving their performance. For example, if the Operating System (OS) task scheduler could dynamically characterize running applications (as memory-intensive or compute-intensive), the task scheduler should consider this characterization into the scheduling decisions to improve the overall system performance. Unfortunately,
1.2 Objectives

Driven by lack of solutions to access the CPU performance monitoring abilities in the Linux kernel, the main goal of this thesis is to design and implement a performance monitoring framework that can be invoked by kernel subsystems, namely the task scheduler. This framework is herein designated as KerMon.

KerMon should be integrated with the OS task scheduler in order to provide higher precision, by only monitoring an application when it is effectively executing on CPU. Moreover, it should assign a higher priority to benchmark applications than the remaining applications. This minimizes interference by background system applications, thus isolating target applications from the unmonitored background applications present on modern OSs.

The task scheduler subsystem is very sensitive to overheads, since it is a facility that is often invoked. For instance, the scheduler tick can be executed every millisecond in order to assess if the currently running task should be preempted. Moreover, the task scheduler instructions are executed in a context where preemption and system interrupts are disabled, thus it can not invoke functions that might sleep. This fact, significantly restricts access to other kernel facilities, namely to `perf_events`, that allows user-space applications to take advantage of the CPU monitoring facilities. Therefore, by aiming KerMon to the restricted task scheduler context, it ensures that it is possible to invoke KerMon anywhere and anytime in the kernel. This fact implies a goal to optimize KerMon in order to have the minimal overhead possible, as well as not to use any functionality that is not available on the task scheduler context.

In the recent years, due to the massification of mobile devices, there is an increasing concern about energy and power consumptions. Thus, a secondary goal is defined in order to accommodate this concern. This goal consists on the KerMon’s synergistic implementation of energy consumption monitoring with the performance monitoring.

In order to assess KerMon, it requires a user-space interface to access the monitoring data. Since this interface has to be implemented anyway, it should be extended so that it is also possible to access KerMon functionality by user-space applications.

The work performed to achieve the referred goals led to the contributions presented in Chapter 3 and the following conference publication:

1. Introduction

aware Roofline Model”, In International Conference on Parallel Processing and Applied Mathematics (PPAM 2013), Springer, Warsaw, Poland, September 2013.
1.3 Dissertation outline

The remainder of this dissertation is organized as follows:

- **Chapter 2 - State-of-the-art**: This Chapter presents an overview of the main concepts and background information necessary to understand the work presented in this thesis. Firstly, it describes the modern multi-core CPU architecture and its facilities for performance and energy monitoring. Then, the inner-working of an OS task scheduler is explained, namely the Linux task scheduler. The different scheduling algorithms of the Linux task scheduler are further described. This Chapter also describes models for performance analysis and characterization of applications. At last, this Chapter presents similar state-of-the-art approaches to the work developed in this thesis.

- **Chapter 3 - The KerMon Framework**: This Chapter introduces the work developed in this thesis: KerMon, a novel framework that allows to take advantage of modern CPU facilities for performance and energy monitoring in kernel-space. Firstly, it provides an overview of the several components that compose the KerMon framework. Then, it exposes some of the difficulties of programming in kernel-space, especially in the task scheduler context. The main content of this Chapter consists on a detailed description of each individual component of the framework, starting from lowest-level kernel-space components (i.e., closest to the hardware) and ending on the user-space tools, that allow a user to take advantage of KerMon. Meanwhile, it also describes the required task scheduler modifications, as well as the methods to reduce the overheads introduced by the KerMon framework.

- **Chapter 4 - Experimental Results**: This Chapter presents an extensive experimental evaluation of the KerMon framework, conducted by relying on the standard SPEC CPU2006 benchmark suite. This Chapter begins by describing the experimental environment. Then, experimental tests are performed using this environment to assess the KerMon overhead impact on the overall system performance and to characterize several SPEC CPU2006 benchmarks by using CARM plots.

- **Chapter 5 - Conclusions**: In this final Chapter, the main conclusions of the presented work are made. It also provides suggestions for future research work.
1. Introduction
State of the art

Contents

2.1 Performance and Energy Counters ................................................. 9
2.2 Task Scheduler ........................................................................ 12
2.3 Linux Task Scheduler ................................................................. 15
2.4 Cache-Aware Roofline Model ...................................................... 24
2.5 Related work ........................................................................... 25
2.6 Summary .................................................................................. 26
Due to advancements of chip manufacturing process and micro-architectural improvements, modern commodity computer systems are becoming true high performance computing machines, capable of delivering a performance in the order of hundreds of GFLOPS [3].

In the past, performance gains were mainly achieved through techniques based on increasing pipeline depth to allow CPUs operating at a higher clock frequency. Unfortunately, these techniques also resulted in a significant increase of the chip power consumption. To overcome this issue and allow for continuous scaling of the overall processor performance, the manufacturers turned to multi-core designs in order to take advantage of SMP.

Figure 2.1: Modern Intel quad-core CPU with Hyper-threading.

Nowadays, modern multi-core processors are equipped with several CPU cores. This leads to an overall architecture composed by different domains as shown in Figure 2.1. These domains refer to different parts of the chip, namely the Cores domain that includes the CPU cores and lower levels of the memory hierarchy such as L1 and L2 caches. A CPU core is an in-chip component that contains the essential execution units such as Arithmetic logic units (ALUs) and Floating-point units (FPUs). When Hyper-threading is active, a single core, by itself, is capable of SMP. Typically, other chip components that do not belong to the core domain are designated as uncore components. As depicted in Figure 2.1 for certain processor architectures, even the Last Level Cache (LLC) can be considered as an example of uncore component, when it is shared between the cores. Other examples of uncore components are the ones that traditionally belonged to the computer chip-set, but are nowadays integrated in the CPU chip, such as Integrated Memory Controller (IMC), Power Management Unit (PMU) and cross-core interconnections, e.g.,
Intel’s QuickPath Interconnect (QPI). Figure 2.1 also presents another region designated as off-core. This region contains essential components sustaining the CPU execution, however they reside outside of the CPU chip. The DRAM is a prime example of an offcore component.

As previously referred, due to technologies such as Hyper-threading, a single CPU core is also capable of Simultaneous Multi-Threaded (SMT). To support this execution concept, some of the core hardware components are replicated and the internal execution units is further optimized in order to allow execution of two threads simultaneously. For example, a quad-core CPU with Hyper-threading, such as the one depicted in Figure 2.1, supports at most 8 instruction streams (4 cores × 2 threads), thus it is able to process 8 threads simultaneously. At the OS level, each instruction stream is designated as a Logical Core (LC), thus to the user, hyper-threading cores are indistinguishable from the real cores.

2.1 Performance and Energy Counters

To fully take advantage of high performance multi-threaded systems, programmers and users should be able to accurately profile demanding applications in order to further optimize their execution. CPU designers, recognizing this need, implemented facilities on modern CPUs for on-the-fly assessment of application performance/characteristics and utilization of the underlying hardware components (both at the processor and the system levels).

The HCs are one of those facilities. They represent a set of special-purpose registers that measure the occurrence of different hardware events, such as clock cycles, retired instructions, branch miss-predictions and cache misses. Thus, HCs can provide low-level information about the on-the-fly performance characteristics of a processor, system and target applications. Modern Intel CPUs allow counting the events in different processor domains (See Figure 2.1), namely:

- **Core events** that are specific for a LC such as FPU, L1 and L2 cache events;
- **Uncore events** that are shared by all cores such as L3 cache events;
- **Offcore events** that occur outside the CPU die such as DRAM events [4].

Starting from Sandy Bridge CPU family, Intel introduced an internal Power Management Unit (PMU) that allows measuring the energy and power consumption of different CPU power domains (planes) or of the whole CPU package. As a result, modern CPU architectures allow performing fine-grained analysis regarding energy and power consumption without the need to attach additional specialized measuring hardware. For Intel micro-architectures, this facility is designated as Running Average Power Limit (RAPL) and it relies on event counters as well as temperature and leakage models, in order to estimate energy consumption. RAPL provides the

---

1In Hyper-threading, a single core only executes two threads simultaneously if there are enough execution resources available according to the requirements of both threads. This behavior is managed by the CPU and it is transparent to the OS. Therefore from the OS and user point of view, both threads are executing simultaneously on a single core.
2. State of the art

means for accessing the energy consumption of different CPU components in different power domains, as follows [6]:

- The Package domain refers to the entire CPU chip;
- All cores and private caches of a CPU are considered as PP0 domain;
- The PP1 domain is composed by the remaining in-chip components (not included in the PP0 domain), e.g., the integrated Graphical Processor Unit (GPU);
- The DRAM interface has its own DRAM domain.

2.1.1 Low-Level Libraries

The HCs are accessed through the Model-Specific Register (MSR) interface. For security reasons, CPUs only allow access to the MSRs by instructions executing on the CPU Ring 0 privilege level. These rings are defined by modern CPUs to restrict access of certain features to instructions executing on a certain level of privilege, where Ring 0 is the highest privilege level. On Linux systems, only the kernel or a kernel module can execute within this privilege level. As a result, any tool that requires taking advantage of MSR-based measuring facilities must rely on a driver or library in kernel-space. The most widely-used low-level libraries are Perfctr and perf_events [7].

Perfctr was the first widely-used kernel patch that allows event monitoring. Since it is not included in the standard Linux kernel, installing Perfctr requires patching and compiling the kernel. These are complex operations that require a deep knowledge of Linux systems and high user access privileges, namely root access. Perfctr was a common solution before the advent of perf_events and currently its development is deprecated [8].

Nowadays, the Linux kernel contains a special subsystem that provides support for performance event monitoring. This subsystem is commonly designated as perf_events (originally Performance Counters for Linux) and it is available since linux 2.6.31 [9]. This subsystem represents a common way to access the performance counters and it does not require patching the kernel or installing kernel modules. It also supports a set of additional features that were previously exclusive to the high-level libraries, such as:

- Event multiplexing, which allows monitoring more event counters than what the hardware can physically support. In brief, current Intel architectures typically provide only 4 configurable MSRs, thus only 4 different hardware events can be simultaneously assessed. To overcome this issue, perf_events automatically schedules different event sets in a round-robin fixed-interval manner and estimated the final results through a statistical extrapolation [7]:
2.1 Performance and Energy Counters

- **Pre-defined events**, that are abstract events that correspond to hardware events, allow perf_events to provide some cross-platform functionality, since these standard events are defined identically, independently on the CPU platform, even if the underlying hardware event is different [10];

- **Periodic sampling**, where a sampling counter triggers an interrupt after a certain number of event counts. The interrupt records the data into a ring-buffer that is accessible from user-space through a mapped memory interface [10];

Beyond counting CPU hardware events, perf_events is also capable of counting software events, i.e., kernel events, such as pages faults and context switches [10]. Recently, since Linux 3.14, perf_events added support for the RAPL interface in order to measure energy consumption [11].

Due to the fact that the perf_events subsystem provides a user space interface through system calls, many user space applications and libraries started to use it in detriment of other drivers. A typical use-case scenario requires: A caller firstly execute the perf_event_open system call in order to setup an event. This system call returns a file descriptor in order to be read by the read system call. Within the perf_event_open system call, the task to be monitored and the targeted LC are also specified. This means that perf_events only allow counting events when the specified task is really executing on the specified LC. In addition, it is also possible to count events for all LCs or for all running tasks. In summary, it is possible to count events for:

- A task (optionally including all children tasks) executing on a specific LC
- A task (optionally including all children tasks) executing on any LCs;
- All tasks (aggregated) executing on a specific LC

Currently, it is not possible to count events of all tasks on all LCs.

2.1.2 High-Level Libraries and Tools

Although it is possible to profile an application using the above-mentioned low-level libraries, developers and users usually rely on high-level libraries for application profiling. High-level libraries provide useful features that often extend the functionality of the low-level libraries, such as cross-platform functionality or counting events of other components besides the CPUs.

Performance API (PAPI) is one of the most popular high-level libraries that provides a consistent interface to access the performance monitoring hardware on CPUs from different vendors, i.e., HCs and energy consumption [12] [13]. Moreover, PAPI can also count software events such as page faults and context switches. Furthermore, it supports functionality extensions, named components, that allow definition of additional events that can be monitored, even from other

---

2 A system call is a function in kernel-space that can be invoked by user-space applications. It is commonly referred in kernel nomenclature as syscall.
2. State of the art

computer components beyond CPUs. For example, the CUDA component provides access to performance monitoring facilities on GPUs supporting CUDA.

The main advantage of PAPI lies in its portability, since it is a cross-platform framework that provides a common monitoring interface for different CPU architectures and different OSs. This is achieved via a unified definition of preset events, i.e., virtual events that provide an abstraction of real hardware events. From the programmer point of view, a predefined preset event is the same for any supported platform, although internally a preset event may be mapped to different hardware event(s) depending on the platform. PAPI denotes intrinsically supported architecture hardware events as native events, thus it also provides a low-level access to the platform. Native events are platform dependent, since different platforms may have different event configurations. As a result, an application that implements PAPI native events does not necessarily inherit the cross-platform portability of PAPI preset events.

In order to support several OSs and CPU architectures, PAPI invokes different low-level libraries depending on the target platform. For x86 (32 or 64 bits) CPU platforms running a recent Linux OS, PAPI relies on perf_events to monitor HCs and energy consumption [14][15]. Since PAPI is an Application Programming Interface (API), a typical use-case scenario requires modification of the target application source code to invoke the PAPI functions.

Another commonly used library is Oprofile, a tool for system-wide profiling [16]. It allows to monitor the entire system, including the kernel and interrupts as well as the execution of a single application. Oprofile provides a cross-CPU platform portability, although it only supports Linux OS. In contrast to PAPI, Oprofile does not require recompilation or modification of the target applications, allowing its usage by non-programmer users. Earlier version of this tool included its own kernel driver for accessing the HCs, which is now deprecated and replaced with the perf_events subsystem [16].

2.2 Task Scheduler

In order to provide an accurate profiling of running applications, the monitoring tools/libraries must closely work with the default OS-level task scheduler. In detail, for a specific monitored application, the library must be able to stop or resume the performance counters when the target application halts or resumes execution, respectively. Therefore, the library must be notified by the task scheduler of the application state changes. While this level of interaction can be achieved for low-level libraries, such as perf_events, the high-level libraries functionality cannot be used by the Linux task scheduler since they are too complex to be invoked on the restricted task scheduler context.

The task scheduler manages the allocation of tasks to the LCs, where a task can be either

---

3 A preset event count value can be extracted directly from a hardware counter or can result from calculations involving one or more hardware counters.
2.2 Task Scheduler

a process or a thread. In other words, it is the task scheduler that decides which task runs, on which LC and for how long it will run.

The task scheduler is the subsystem responsible for the multitasking functionality of modern OSs. This functionality allows the execution of several applications concurrently even on systems with a single LC. The task scheduler achieves this functionality by executing a single task for a short period of time, then it pauses the task execution in order to perform some other tasks, and it eventually resumes the execution of the paused task for another short period of time and then it pauses the task again. This procedure is repeated until the task finishes execution. Essentially, the task scheduler is performing time based multiplexing on the available LCs. Since the scheduling time interval is very short and imperceptible to human senses (in the order of milliseconds), these tasks seem to be executing simultaneously even on single threaded systems. The referred task switching mechanism is commonly designated as a context switch, since the task scheduler needs to perform a set of operations to store the register values for a paused task, as well as to restore those values for the resumed task.

In general, task schedulers can be classified as cooperative or preemptive. The cooperative task scheduler only performs a context switch when the currently running task voluntarily yields execution. In case of an application malfunction or a bad intentioned application, it may cause an unresponsive system, especially on single threaded systems. The preemptive task scheduler does not suffer from this issue, since it can force the task to halt its execution. A forced context switch by the task scheduler is designated as preemption. In fact, preemptive task schedulers also allow a task to voluntary yield the LC for example when it is waiting for resources. Modern OSs, such as Linux, rely on preemptive task schedulers, thus all scheduling examples presented in this dissertation address the functionality of the preemptive task schedulers.

For a classic task scheduler, the preemption decisions are driven by the definition of a time interval designated as epoch or scheduling period. The epoch time is then distributed among each runnable task, i.e., the tasks that are capable of running immediately and that are only waiting for attribution of CPU time. An epoch slice attributed to a task is designated as time-slice. During an epoch, a task can not execute longer than its time-slice. To enforce this mechanism, the task scheduler setups system interrupts in order to periodically execute a set of predefined operations (function). This system interrupt occurrence is referred as scheduler tick. On the scheduler tick, the task scheduler updates and checks the current running task statistics, and preempts the current task if its time-slice is depleted. Preemption can also occur when a higher priority task becomes runnable. On Linux OS, the user can influence a task time-slice by changing the nice level of a task.

Figure 2.2 shows an example of a timeline of the task T on a single LC where all tasks have the same priority. For brevity, the continuous interval where a task is assigned and executing on a LC

\footnote{In the context of this dissertation, the hardware notion of task execution is applied, i.e., whenever it is referred that a task is running or executing, it is assumed that the task instructions are currently being executed on the CPU.}
2. State of the art

Figure 2.2: Scheduler timeline of task $T$ on a single LC. The task $T$ is created on A and exits on I. The gray area represents the time when the task was running, while the dotted lines represent scheduler ticks.

is denominated herein as task session (see grayed areas in Figure 2.2). As it can be observed, the task is firstly created and inserted into a queue (A). This queue is referred as a runqueue, since it only contains runnable tasks. When a context switch occurs, the scheduler picks the first task from the runqueue and executes it. For a hypothetical task $T$ in Figure 2.2 this is represented with (B), where the scheduler executes task $T$ and Session 1 begins. In case that task $T$ requires some operations that are not immediately available, such as Input-Outuput (C), the resource driver may halt the task $T$ and insert it into a wait queue. In this case, the scheduler is invoked to execute the other tasks (C), thus ending the Session 1.

A wait queue contains tasks that are waiting for an event. In contrast to the runqueue, the wait queues are not managed by the scheduler. They are managed by the driver responsible for the event on which the task is waiting. For example, if a task halts due to waiting for data from a file, the filesystem driver is responsible to keep the task information in its own wait queue.

When the resource becomes available, the task $T$ is moved from the wait queue to the runqueue (D). Eventually, the task scheduler selects again the task $T$ (E). When the scheduler tick function detects that the time-slice of task $T$ is exhausted (F), the scheduler preempts task $T$ to allow other tasks to execute. Therefore, task $T$ will not run until the current epoch finishes and a new one starts (G). In the new epoch, the task $T$ resumes execution (H) until it exits (I). The task scheduler is invoked again to pick a next task to be executed.

Besides managing multitasking functionality, in SMP systems, the task scheduler is also responsible for making decisions on which LC a task will run.
2.3 Linux Task Scheduler

On Linux systems, newly created tasks start executing on the same LC as the parent task. Since all processes are descendents from the init process\(^5\), without any other mechanism, all tasks will be running on the same LC as the init process, thus leaving the remaining LCs idle. Therefore, the task scheduler implements a mechanism, designated as load-balancing, that distributes the tasks among the LCs in order to balance the computational load between them. For this, the scheduler performs task migrations, which refer to the procedure of changing the LC that is currently assigned to a task. On Linux, load-balancing occurs in the following situations\(^6\):

- When a task is awaken from a waiting state, it can be assigned:
  - To the same LC where it was previously executing, in order to take advantage of cache locality;
  - To the LC with the least load;
- When an exec system call is invoked or a task is created, the task is migrated to the LC with the least load;
- When a LC is about to become idle, the scheduler tries to pull tasks into this LC;
- By a periodic trigger, to rebalance the utilization of all LCs.

Since the scheduler controls CPU allocations, the scheduler context is very restrictive, where the most scheduler operations are protected against preemption. This means that while the scheduler is executing, it locks the LC where it is running until it finishes execution. Furthermore, the scheduler context executes with the interrupts disabled, thus no operation that relies on interrupts can be used inside the scheduler.

Linux has only one task scheduler, although it supports several scheduler algorithms, through the concept of scheduling class. Each scheduler class is responsible to maintain, track and manage the execution of runnable tasks, thus each class has its own runqueue(s). The scheduler communicates with each class through a set of function pointers, defined by the structure sched_class. A scheduling class must create an instance of this structure and implement the handlers for those function pointers. It should be noticed that not all function pointers are required to be implemented.

In order to better illustrate the main functionality of the task scheduler, the previously referred example in Figure 2.2 is followed herein by Figure 2.3.

---

\(^5\)The init process is the first process to be executed on a Linux OS boot in order to control the boot procedure after the kernel was loaded. It is invoked directly by the Linux kernel and it lives until the OS is shutdown.

\(^6\)The exec system call replaces the program executing on the current process.
On situation (A), the task \( T \) is created and inserted into a runqueue. The scheduler invokes the `task_fork` handler in order to notify the task \( T \) scheduling class of its creation. Then, it invokes the `enqueue_task` handler of the respective scheduling class in order to insert task \( T \) into the corresponding runqueue. When the \[\text{LC}\] is freed, the scheduler invokes the `pick_next_task` handler that picks task \( T \) to execute (B). When the task \( T \) needs to wait (C), thus stops being runnable, the scheduler first invokes the `dequeue_task` in order to instruct the task's scheduling class to remove it from the runqueue. Next, the scheduler invokes the handler `put_prev_task` to inform the task's scheduling class that the task is halted. When the task \( T \) awakes (D), the scheduler notifies the task's scheduling class through `task_waking` and `task_woken` handlers before invoking again the `enqueue_task` handler. Then, task \( T \) is scheduled to execute (E) following the same procedure described for (B).

In brief, on scheduler tick, the `task_tick` handler is invoked for the current running task's scheduling class, and the preemption decision is made. Hence, on event (F) the `task_tick` handler decides to preempt the current task and it notifies the task scheduler. Then, the task scheduler halts the task and invokes the handler `put_prev_task`. In contrast to event (C), the
2.3 Linux Task Scheduler

dequeue_task handler is not invoked since the task $T$ remains runnable.

The Linux task scheduler considers event (G) as a regular scheduler tick, since only the scheduling class has the notion where an epoch starts. Task $T$ is scheduled to execute (H) in the same way as described for event (B). When the task exists (I), the scheduler demands the removal of the task $T$ from the runqueue through the dequeue_task handler. Then, it notifies the task's scheduling class of the exit of task $T$ through the task_dead handler.

As it can be observed in this example, the scheduler is not aware of any details regarding the scheduling algorithm. In fact, the whole scheduling functionality is encapsulated within each scheduling class. Therefore, with this abstraction layer, the Linux task scheduler is able of supporting several scheduling algorithms simultaneously.

Since Linux 3.14, 3 different scheduling classes are officially supported [19], namely:

- **A Deadline** class that implements a scheduling algorithm composed by Earliest Deadline First (EDF) and Constant Bandwidth Server (CBS) algorithms [19];
- **A Real-Time (RT)** class that implements ‘First In - First Out’ (FIFO) and Round-Robin scheduling algorithms;
- **The Fair** scheduling class that applies the CFS algorithm, which is also the default class.

The assignment of tasks to the scheduling class is performed through the task's scheduling policy. Each scheduling policy is associated with only one scheduling class, although a class can be associated with multiple policies, as can be seen in Table 2.1. It is worth noticing that Linux includes additional scheduling classes, but they are reserved for internal kernel usage and they cannot be used directly.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Scheduler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHED_NORMAL</td>
<td>Fair</td>
<td>Also known as SCHED_OTHER, it's the default policy for the regular tasks.</td>
</tr>
<tr>
<td>SCHED_BATCH</td>
<td>Fair</td>
<td>Tasks doesn't preempt as often as regular tasks, allowing tasks to run for a longer time.</td>
</tr>
<tr>
<td>SCHED_IDLE</td>
<td>Fair</td>
<td>The task will only run when the LC is idle. This policy is currently disabled due to a priority inversion problem.</td>
</tr>
<tr>
<td>SCHED_FIFO</td>
<td>RT</td>
<td>A FIFO based scheduling policy.</td>
</tr>
<tr>
<td>SCHED_RR</td>
<td>RT</td>
<td>A round-robin based scheduling policy.</td>
</tr>
<tr>
<td>SCHED_DEADLINE</td>
<td>Deadline</td>
<td>An EDF and CBS based scheduling policy.</td>
</tr>
</tbody>
</table>

Table 2.1: Scheduling policies and classes relationship [1].

When selecting a task to be executed, the scheduler iterates through the scheduling classes. For each class, it requests a task in their runqueue to be executed, as shown in Figure 2.4. The scheduler first request a task from the Deadline scheduling class through the pick_next_task handler. If the handler returns a task, the scheduler executes it. Otherwise, it repeats the same procedure for the RT scheduling class. If there are no runnable RT tasks, the scheduler requests
2. State of the art

Figure 2.4: Scheduling class iteration to select a task to be executed.

the task from the Fair scheduling class. Finally, if the default scheduling class (Fair) returns no tasks, the CPU becomes idle. The scheduler performs this selection operation whenever the kernel function schedule\footnote{More precisely, the scheduler is executed by the auxiliary function \texttt{\_\_schedule}.} is invoked. This occurs in the following situations:

- Current task voluntarily yields execution, e.g., by invoking the system call \texttt{sched\_yield};
- Current task is inserted in a wait queue, i.e., it is waiting for an event or a resource;
- Current task exits, i.e., its execution is finished.
- A previously disabled\footnote{More precisely, the scheduler is executed by the auxiliary function \texttt{\_\_schedule}.} became enabled and ready to execute tasks;
- A preemption condition is met, e.g., when the time-slice of a task is exhausted.

Figure 2.5: A task’s state through the its lifecycle.

As it can be noticed, the scheduler is invoked mainly when the task state changes. A task keeps this information in its state structure property. Figure 2.5 shows how the most important
states relate with the lifecycle of a task. When a task is created (A), it is set as a runnable task (TASK_RUNNING) (B). It waits in the runqueue until the scheduler selects it to be executed (C). A task maintains the TASK_RUNNING state whether it is in the runqueue or it is really executing on the CPU since in both cases, the task is considered runnable. The task scheduler can be invoked to perform a context switch in the following conditions:

- The task is preempted and it returns to the runqueue, without any state change (D). This is also the path of a task that voluntarily yields the CPU by maintaining its runnable state;
- The task needs to wait for resources to become available or for an event to occur (E). In this case, the task sleeps until the waiting condition is met, and then it returns to the runqueue (B). While the task is sleeping, it is kept into a driver's wait queue and it changes its state into one of the following states:
  - TASK_INTERRUPTIBLE, designates a task that can be waken up upon an interrupt for processing signals or explicitly by the driver;
  - In contrast, a task with the state TASK_UNINTERRUPTIBLE ignores interrupts and can only be waken up with an explicit wake-up call;
- The task finishes execution, and its state is changed to TASK_DEAD (F). In fact, after releasing its resources, the task is finally deleted.

### 2.3.1 The Deadline scheduling algorithm

When the task scheduler iterates scheduling classes in order to pick a task to execute, the Deadline scheduling class is the first one to be queried. This order provides an advantage to Deadline tasks, since they are always executed in detriment of tasks belonging to other classes. Hence, if the Deadline class continuously provides tasks to be executed, no other tasks will be executed. Therefore, it is implicitly assumed that Deadline tasks have the highest priority in a Linux system.

The Deadline scheduling class is the most recent scheduling class inserted in the official Linux kernel (available since Linux 3.14). The Deadline scheduling class implements the Earliest Deadline First (EDF) algorithm [19], which provides Linux with the capability of managing tasks with timing constraints. As suggested by its name, when selecting a task to execute, the EDF algorithm picks the task that is closest to its deadline.

The Deadline scheduling class merges the EDF algorithm with the Constant Bandwidth Server (CBS) algorithm in order to ensure temporal isolation between tasks, i.e., to guarantee that a task's ability to meet deadlines is not affected by other tasks.

CBS characterizes tasks by two time constraints: budget and period, where the period is also considered as the task deadline. In detail, CBS schedules a task in order to execute it exactly
2. State of the art

This imposes a restriction in terms of the number of concurrently running tasks in this class, where \( \sum_{tasks} \frac{\text{budget}}{\text{period}} \) cannot exceed 1 for a single threaded system. If this value is higher than 1, the deadlines will not be met [20].

2.3.2 Real-Time Scheduling Class

The \texttt{RT} scheduling class is the second class queried by the task scheduler, when selecting a next task to be executed. Therefore, \texttt{RT} tasks will only run when there is no runnable task from the Deadline scheduling class. The \texttt{RT} scheduling class implements two different real-time fixed-priority preemptive algorithms, namely: ‘First In - First Out’ (\texttt{FIFO}) and Round-robin. The behavior of those algorithms is defined by the Portable Operating System Interface (POSIX) standard [21], and both adhere to the following set of basic rules:

- The task with the highest priority is selected;
- When several tasks have the highest priority, the task that was inserted first into the runqueue is selected;
- When a task becomes runnable, it is inserted at the end of the runqueue;
- While a task is running, if a higher priority task becomes runnable, the current task is immediately preempted and the higher priority task starts executing;
- When a task is preempted by a higher priority task, it does not lose its position in the runqueue.
- When the class of a task is set to the \texttt{RT} scheduling class or when its priority is changed, this task is inserted on the beginning of the runqueue, preempting the current task if the latter has equal or lower priority.

In fact, the \texttt{FIFO} algorithm applies only the above rules. However, these rules alone may cause \texttt{CPU} starvation, since a lower priority task will never run if there are runnable higher priority tasks. Furthermore, \texttt{CPU} starvation can also occur among tasks with the same priority, since the currently running task can indefinitely hold the \texttt{CPU}.

The Round-Robin algorithm solves the \texttt{CPU} starvation among the tasks with the same priority by adding a rule: A task is only allowed to execute continuously for a certain time (time-slice). After the task’s time-slice is exhausted, the task is reinserted at the end of the runqueue.

As it can be observed, by defining the time-slice, the Round-robin algorithm does not allow that a task executes indefinitely while there is another runnable task with the same (or higher) priority in the runqueue [22].

\footnote{One can notice that the concepts of \texttt{budget} and \texttt{period} are very similar to classical concepts of time-slice and epoch, respectively.}
2.3.3 **CFS** Scheduling algorithm

The Fair scheduling class is the last class that the task scheduler checks in order to pick the next task to be executed. This means that a task of this class is executed only if the other classes do not provide tasks to the task scheduler. It also means that if the Fair scheduling class does not provide any task, the current LC will become idle.

The Fair scheduling class implements the Completely Fair Scheduler (CFS) algorithm. This algorithm tries to replicate a model of an ideal, precise multi-tasking LC. In this model, an ideal LC can execute all tasks at the same time, dividing its processing throughput by the concurrent tasks [1].

CFS emulates this ideal model by dividing the LC execution time among all runnable tasks. The CFS has a slightly different notion of the classical time-slice. Instead of using time-slices, it introduces the concept of virtual runtime (\(vruntime\)) of a task which corresponds to the actual runtime of a task, normalized by the total number of running tasks. Using this concept, the scheduler calculates the ideal runtime, i.e., the expected CPU time that the task should have gotten.

On CFS's `pick_next_task` invocation, it chooses the task with the smallest virtual runtime. This criteria allows that a task that has less execution time (smaller \(vruntime\)) is selected so that it can achieve the execution time of the remaining tasks (with bigger \(vruntime\)). Thus, on normal conditions, all tasks should have similar virtual runtimes, thus approaching reality with the ideal multi-tasking LC model.

![Figure 2.6: Time-slice calculation procedure when a task is inserted into a CFS runqueue.](image)

Figure 2.6 presents how CFS calculates a task's ideal runtime (time-slice) when it is inserted into a runqueue (1). This calculation only considers tasks that are assigned to the same runqueue. In order to calculate the time-slice, the CFS first determines the epoch length by using `sysctl` runtime variables (2). The first major variable is the `sched_latency_ns` that defines the maximum waiting time for executing a runnable task, i.e., the scheduling epoch or period. The second major variable is `sched_min_granularity_ns` that defines the minimum time that a task can run without
being preempted. Therefore, \(\text{sched\_latency\_ns} / \text{sched\_min\_granularity\_ns}\) returns the maximum number of time-slices of an epoch (slots). If there are more tasks in the runqueue (\(\text{nr\_running}\)) than slots (3), the epoch is expanded so that it contains \(\text{nr\_running}\) slots of length \(\text{sched\_min\_granularity\_ns}\) (4). 

CFS does not support task priorities, i.e., all tasks execute with the priority of 0. However, it supports task weight, that allows a user to assign more CPU time to some tasks, i.e., the bigger the task’s weight comparing with the remaining tasks, bigger time-slice is assigned. Thus, a task’s weight is one of the major factors in time-slice calculation (5). The weight is indirectly controlled by users through the nice tool [23] that manipulates the nice level of processes. The nice level ranges from -20 to 19, with the default value of 0. A task’s weight is roughly equivalent to \(1024/1.25^{\text{nice}}\) (6). Hence, the lower nice level, more weight is assigned to a task and therefore it will have a bigger time-slice, i.e., the task is less nice to other tasks. The nice level is designed to implement the concept that a nice level corresponds to about 10% of time-slice [24]. Thus, a task with nice level of -1 should execute 10% more than a task of nice level 0. The epoch is split between the tasks in the runqueue according to weight. If all tasks have the same nice level, they will also have the same time-slice length. This weighted mechanism specifies that the time-slice of a task corresponds to the task weight share among all tasks in the runqueue, thus

\[
\text{slice} = \frac{\text{Task weight}}{\sum_{\text{runqueue\_tasks\_weight}}} \times \text{epoch} \tag{7}
\]

The CFS optionally supports task groups, where tasks can be grouped according to a certain criteria, such as the user that invoked the tasks. By using a structure denominated as entity that can represent either a task group or a task, the scheduler is able to handle tasks groups indifferently from tasks. CFS first splits the CPU time between the groups and ungrouped tasks. Then, it splits a group time-slice between its tasks. In practice, this operation is performed by applying another factor on the already obtained time-slice. Therefore, tasks that belong to groups, have their time-slice further reduced by \(\frac{\text{Group weight}}{\sum_{\text{runqueue\_tasks\_weight}}}\) (8).

Figure 2.7 depicts the lifecycle of a task in CFS algorithm. When a task is created (1) and becomes runnable (2), it is inserted on a runqueue and the time-slice is calculated. When the scheduler picks this task among the runnable ones (3), it also removes the task from the runqueue and assigns it to the LC to be executed. While the task is running (4), a scheduler tick periodically occurs, briefly halting the current task in order to execute the scheduler instructions. The scheduler tick updates the task’s statistics in order to be used for scheduler decisions (5), such as vruntime. If the vruntime value is bigger than the time-slice, the task is preempted (7), halting its execution and the scheduler is invoked (8). Otherwise, the current task resumes execution. A task can also be halted for other reasons such as waiting for resources or events (8). A task path after (9) depends on the reason of scheduler invocation, namely:

- Preemption- When the task is reinserted into the runqueue, keeping its runnable state (2);
- Waiting for an event or resource- Task is inserted into a wait queue managed by the event or resource driver (10). When the event is dispatched or the resource is available (11), the
2.3 Linux Task Scheduler

Figure 2.7: Task Lifecycle on CFS.

- Task becomes runnable and is transferred from the wait queue into a runqueue (2);
- Task ends its execution- The task is destroyed, ending its lifecycle (12).

Figure 2.8: Simplified example of a runqueue implemented by a Red-black tree.

When CFS requires a task from the runqueue to be executed (Event (3) in Figure 2.7), it selects the task with the smallest vruntime. In order to minimize overheads, the runqueue is implemented as a search tree data structure sorted by the tasks vruntime. More precisely, it is a red-black tree as the one depicted in Figure 2.8. The red-black tree is a self-balanced binary search tree where each node has two children sub-trees. The left sub-tree only contains tasks with a smaller vruntime than its parent, while the right sub-tree contains tasks with a bigger or equal vruntime than its parent. Thus, when CFS selects a task to execute, it just picks the leftmost
2. State of the art

That implies a decision in the performance order of $O(1)$, while the insertion performance is in the order of $O(\log_2(n))$ [25].

2.4 Cache-Aware Roofline Model

As previously referred, modern CPUs contains multiple processing cores in order to increase processing performance. On recent Intel CPUs, each core has its own private set of caches (L1, L2), while other memory levels (L3, DRAM) are shared among the cores.

Since data accesses and computation operations are performed in parallel, the execution is limited either by the core’s computation resources or by the memory subsystem capabilities. For instance, if a task performs a lot of memory operations and only a small amount of computations, the memory subsystem will stall the execution. In this scenario, the memory subsystem is the system bottleneck, thus, it does not allow to reach the peak performance of the core’s computational resources. Based on this observation, the Original Roofline Model (ORM) [26] shows the attainable performance of a multi-core architecture by relating its peak Floating Point (FP) performance $F_p$ (in flops/s) with the theoretical bandwidth of a single memory level, usually DRAM (in DRAM bytes/s). In order to perform this relation, it is defined a new term: Operational Intensity (OI), that is the number of Floating point Operations (FLOPs) executed per byte of DRAM traffic (in flops/DRAM bytes). However, since memory is composed of several hierarchic levels, this model cannot fully describe the behavior of modern applications and architectures by individually analyzing the behavior of each level.

In practice, the accesses to different memory levels can not be decoupled, since the data must traverse the whole memory hierarchy before computations are performed. The recently proposed CARM [27] considers these effects and the complete memory hierarchy. In summary, it models the performance upper-bounds of multi-core architectures having into account the different memory levels, in a single plot. In order to achieve this, the CARM considers performance, $F(\phi)$, and bandwidth, $B(\beta)$, as continuous functions of performed FLOPs $\phi$ and transferred bytes $\beta$ at different memory levels. Contrary to the ORM, CARM perceives information from the point of view of the core, allowing it to normalize the information. Therefore, in CARM, the OI (I in flops/bytes) is uniquely defined and the attainable performance $F_a(I)$ of the architecture is expressed as follows:

$$F_a(I) = \frac{\phi}{T} = \min\{B(\beta) \times I, F(\phi)\}, \quad T = \max\{\beta, \phi\}, \quad I = \frac{\phi}{\beta}$$

Equation 2.1 states that $F_a(I)$ is limited either by the memory bandwidth or by the computational performance. Indeed, since memory transfers and computations overlap, the overall execution is dominated either by the time to transfer the data ( $\beta/B(\beta)$ ) or by the computation time ( $\phi/F(\phi)$ ).

Figure 2.9 shows the CARM for a quad-core Intel 3770K processor. As it can be observed, $F_a(I)$ is bounded by:
2.5 Related work

At the date of this writing, there is not a clearly identical work to the one developed in this thesis. The most similar work can be split in 2 categories, namely:

- **Kernel drivers** that allow user-spaces applications to monitor HCs such as the already referred perf_events [10], Perfctr [8] and OProfile [16]. Those facilities were used by several profiling tools such as the well-known user library PAPI [28]. A recent kernel driver is SchedMon, that provides a kernel module to monitor performance, energy consumption, and to trace functions [29].
2. State of the art

- **Scheduler modifications** to improve overall system performance by overriding the default task scheduler. Current state-of-the-art approaches amend scheduler decisions by relying on HC\textsuperscript{s} to classify running tasks and describe utilization of system shared resources \cite{30} \cite{31} \cite{32}. However, these approaches must provide their own drivers to access HC\textsuperscript{s} in the kernel, since the previously described drivers serve for user-space applications.

The work developed in this thesis provides a framework to access HC\textsuperscript{s} and the RAPL interface in kernel-space, especially in the scheduler context. Therefore, it is a low-level framework that aims low latency and negligible overheads. Its integration within the kernel scheduler provides an highly accurate performance information for the profiled tasks. It also integrates state-of-the-art application characterization mechanisms based on HC\textsuperscript{s}, i.e., CARM and ORM. These mechanisms allow to identify which system component is limiting an application performance, i.e., the bottleneck.

2.6 Summary

This chapter describes the state of the art of performance and energy monitoring tools, especially on multi-processing system, that adds complexity when analyzing a system/application performance. Thus, nowadays CPUs have some facilities, such as HC\textsuperscript{s} and RAPL, that provide performance event counting and energy consumptions readings, respectively. There are performance monitoring tools that already benefit from these new facilities, including the low-level in-kernel perf\_events. perf\_events has a reduced feature set, thus, in order to access additional features, a developer must use a high-level user-space tool such as PAPI\textsuperscript{\textregistered} and Oprofile.

The scheduling decisions impacts performance, thus, in this state of the art, it is analyzed the Linux task scheduler mechanism and its algorithms, implemented through scheduling classes: Deadline, Real-Time and Fair. The default CFS algorithm is especially analyzed, that is implemented by the Fair scheduling class and it tries to replicate an ideal and unrealistic system where every application is executing at the same time, by dividing the CPU performance between them.

The cache-aware roofline model is also described, a novel manner to characterizing performance of real-world applications by considering the different levels of memory hierarchy on a multi-core architecture.

This chapter ends with a comparison of the related work to the work proposed by this thesis, where is concluded that although there is similar work, none of them meets the requirements for the proposed work.
3

The KerMon framework

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Overview</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Considerations of programming on the task scheduler context</td>
<td>30</td>
</tr>
<tr>
<td>3.3 Low Level access to Hardware Counters</td>
<td>31</td>
</tr>
<tr>
<td>3.4 Event Logger</td>
<td>39</td>
</tr>
<tr>
<td>3.5 KerMon scheduling classes: Scheduler++ and CFS++</td>
<td>46</td>
</tr>
<tr>
<td>3.6 User-Space tools</td>
<td>51</td>
</tr>
<tr>
<td>3.7 Summary</td>
<td>54</td>
</tr>
</tbody>
</table>
3. The KerMon framework

3.1 Overview

Nowadays, widely-used OSs (such as Windows and Linux) do not generally take advantage of the performance and energy monitoring facilities in modern CPUs, to perform scheduling decisions or to characterize the execution of running applications. With this additional data, it is possible to deeply assess how different execution phases of modern applications are capable of exploiting the capabilities of the underlying hardware, as well as to detect the potential execution bottlenecks in different stages of the processor pipeline. As a result, the task scheduler can provide better decisions. This fact lead to the development of the work presented on this dissertation: KerMon, a framework that introduces performance and energy awareness to the kernel subsystems, namely the task scheduler.

Although it may seem counter-intuitive, implementing this functionality with the already existing Linux monitoring facilities it is not an easy task. For example, perf_events requires substantial modifications to be made, in order to allow its invocation by kernel subsystems, since perf_events is designed for user-space applications. Furthermore, additional complex modifications are required to allow its invocation by the task scheduler, due to the limitations of task scheduler instructions that are executed with disabled interrupts, thus it is not possible to invoke sleeping functions. Moreover, since the task scheduler executes every few milliseconds, its functions must be kept simple, fast and with a very low overhead. Even if perf_events is adapted to be invoked by the scheduler, it will still add a significant amount of overhead due to functionality that the scheduler does not generally require. Thus, perf_events, even if modified, is not suitable for providing performance and energy monitoring information to the task scheduler. Therefore a simpler, faster and highly optimized solution for the task scheduler was designed and implemented, i.e., the KerMon framework. Although the main KerMon functionality is implemented in the task scheduler, it can be used with other kernel facilities. In fact, it can even be controlled by user-space applications to perform customizable performance and energy monitoring of applications.

KerMon is composed by a set of components, positioned in the overall computer systems. Figure 3.1 depicts a system with KerMon components represented by numbered round-corner boxes, namely: Raw Events (1), Event Mux (2), Event Logger (3), Scheduler++ (4), libSched++ (5) and kernelCount (6).

As it can be observed, KerMon main components reside in kernel-space. The Raw Events component (1) also resides in the kernel-space, although it is adjacent to the hardware level, since it provides a low-level and negligible overhead access to the CPU’s hardware facilities, such as MSR’s, HC’s and RAPL.

Since the number of available HC’s is very limited, the Event Mux component (2) is introduced to provide event multiplexing functionality, i.e., simultaneously count more events than the
3.1 Overview

Figure 3.1: This Figure depicts KerMon components, represented by numbered round-corner boxes, in an Intel/Linux computer platform.

physically supported by the underlying hardware. dividing the total number of events to be monitored into a minimum amount of event sets, such that all events within a single event set can be monitored simultaneously.

The Event Logger (3) is another kernel-space component. It communicates with the Event Mux component in order to setup performance monitoring events, fetch and compile their values into samples. Each sample contains the monitored performance counters values, energy consumption readings, timestamps, CPU id and Process IDentifier (PID). Then the compiled samples are stored into the Event Logger buffers, namely: Sliding Window and Ring Buffer.

In kernel-space, KerMon is integrated within the task scheduler. This is accomplished through the addition of the new Scheduler++ scheduling class (4). This new class is based on the existing Fair scheduling class with the CFS algorithm (see Section 2.3.3) and it includes several modifications.

A scheduling class performs scheduling decisions when invoked by the scheduler tick or when a task voluntary yields execution. In these scenarios, the scheduler must decide which task to execute next or if task migration should be performed. The procedure of switching running tasks on a LC is designated as context switch. The main purpose of Scheduler++ is to introduce
modifications to invoke the Event Logger on certain scenarios, as depicted in Figure 3.1, namely:

- A scheduler tick invokes Event Logger periodically (A) to:
  - Fetch performance counter values, compile them into a sample and store it;
  - Switch the event group that is currently counting;

- Alternatively, if context switch occurs (B), the Event Logger must be invoked in order to:
  - Take a sample and halt counting, if the halted task’s class is Scheduler++;
  - Configure performance events, if the class of the next task to execute is Scheduler++;

The secondary purpose of Scheduler++ is to isolate benchmark applications from the remaining background system applications, that otherwise may impact the benchmarking values.

Figure 3.1 also depicts user-space KerMon components, such as the user-space library libSched++ (5), that mainly provides a wrapper for the system calls, for instance, to retrieve the Event Logger samples (C), or to define an application scheduling class (D).

Another user-space component is kernelCount (6), a helper tool that simplifies application profiling by configuring the events and by executing the target application on the Scheduler++ scheduling class. While the target application is executing, kernelCount retrieves the performance and energy readings (samples) from the Event Logger through libSched++, and stores them into a file. With this information, it is possible to perform characterization of a target application and system through several analysis models, namely, the CARM that models the performance upper-bounds of multi-core architectures having into account the different memory levels, in a single plot. Thus, by inspecting the resulting plot, performance bottlenecks can be detected in order to apply optimizations that avoid those bottlenecks.

3.2 Considerations of programming on the task scheduler context

In this section, it is discussed the hazards, difficulties and solutions encountered in order to implement KerMon. Some of the difficulties are due to the Linux kernel complexity and its preemptive nature, where most kernel functions can be preempted. Furthermore, on a SMP system, the Linux kernel executes several tasks in parallel, including its own internal tasks. As typical of parallel programming, unaccounted race conditions may lead to sporadic and unpredictable system states. While debugging parallel user-space applications is challenging, debugging the Linux kernel, namely the task scheduler is specially arduous due to:

- The Linux kernel complexity, with a significant amount of parallel events that can severely difficult or make it impossible, to replicate a kernel state;
3.3 Low Level access to Hardware Counters

- The compiler, that optimizes the kernel code, thus the resulting instructions may not correspond to the source code;

- Complex setup for live debugging, that requires special hardware or emulators;

- Faults that completely hang the system, without outputting any information about the bug location.

- The task scheduler functions are executed frequently, for instance, the scheduler tick can be set to execute every millisecond. This implies that any additional overheads in these functions will significantly impact system performance. Thus, the KerMon development have a special concern about overheads and performance.

Live debugging is very common and easily performed on user-space applications, where the developer can execute instructions step-by-step and analyze the variables. Due to the above reasons, kernel live debugging is significantly more complex and rarely is advantageous when compared with the simple debugging through messages prints by the printk function, that works similarly as the user-space printf function.

There are some exceptions where kernel instructions can not preempted, such as the task scheduler functions. A very significant part of KerMon executes on the task scheduler, which context is very restricted, e.g., preemption and interrupts are disabled. This means that it is impossible to use sleeping functions such as mutexes. Thus, this fact restricts the invocation of most kernel features and facilities. For instance, as shown in Figure 3.2 the kernel in-built error handling functions are not prepared for the scheduler context. In this case when a fatal error (kernel panic) occurs, a second kernel panic is thrown since the schedule function is invoked already inside the task scheduler context. This is a real-world example of how the task scheduler context is very sensitive, without any tolerance for errors.

3.3 Low Level access to Hardware Counters

Unfortunately, the Linux perf_events facility that provides access to the hardware performance counters, is one of the kernel facilities that is impossible to take advantage in the task scheduler context. In an attempt to access the perf_events functionality on KerMon, some modifications of this facility were performed. However, this path revealed to be unfeasible and instead, the Raw Events component was developed from scratch.

3.3.1 Raw Events

Raw Events is a library of specially developed functions that provides low-level access to the HCs in kernel-space, mainly in the scheduler context. The developed library is a very fast, simple and light-weighted approach to introduce performance and energy monitoring in the OS.
3. The KerMon framework

![Illustration of a double kernel panic](image)

Figure 3.2: Illustration of a double kernel panic. A fatal error (page fault) occurred inside the task scheduler context. The page_fault handler kills the task and invokes the scheduler. Since it was already running in the scheduler context, the scheduler detects this situation and provokes a second kernel panic.

Raw Events is the KerMon component closest to the hardware. It provides a raw access to the Model-Specific Registers (MSRs), that, just as the name hints, there are differences among CPU models. The work presented in this dissertation targets Intel x86 (64 bits) and Linux platform, Raw Events is designed to take advantage of this platform: It supports energy monitoring on all CPUs that provide a RAPL facility. All Intel CPUs since the first Intel Core generation are supported for the remaining features of this work such as performance event monitoring.

Raw Events is the only KerMon kernel-space component that is dependent on the hardware model, while the remaining components are hardware independent. Figure 3.4 shows the basic functionality of the Raw Events when dealing with the performance HCs. As it can be observed, this functionality is attained via 4 specifically developed functions, namely: `event_setup`, `enable_event`, `reset_counter` and `clear_event`. A performance event counter can be configured by invoking the function `event_setup`. For this one needs to specify which event to monitor and which counter. The counter number is an integer where 0 corresponds to the first HC, 1 to the second and so on, while the event value is a concatenation of `Umask Value + Event Num` as defined in [4]. The function `event_setup` provides the option to start counting immediately or only
3.3 Low Level access to Hardware Counters

![Diagram showing hardware components](image)

Figure 3.3: Top part of Figure 3.1 where it shows that the Raw Events component connects KerMon to the hardware.

![Flowchart](image)

Figure 3.4: Flowchart representing a simple usage model of Raw Events to set and read a performance counter.

after the invocation of the enable_event. The enable_event is a function that allows the client to enable or disable counting. In brief, the enable_event does not reset counter values, since for these purpose, reset_counter must be called. After setup, the function read_counter is invoked to obtain a counter value. This function can be invoked repeatably, e.g., to perform periodic sampling. After monitoring a configured performance event, the counter can be stopped and its value cleared by invoking the clear_event function.

In detail, each function present in Figure 3.4 has variants for each type of event that modern Intel CPUs supports [4], namely:

- The general counters can be configured to count core and offcore events. Core events are events that occur on the CPU core domain. Correspondingly, the offcore events occur on the offcore domain. Recent Intel CPUs contain 4 general counters per LC and they support at most 2 offcore events to be counted simultaneously;
3. The KerMon framework

- The **fixed counters** can not be configured beyond being enabled or not. This means that one fixed counter can only count one hardware-fixed core event. Recent CPUs contain 3 fixed counters per LC.

- The (general-purpose) **uncore counters** can be configured to count uncore events.

- The **fixed uncore counters** are similar to the already mentioned fixed counters, however they count fixed uncore events. Modern Intel CPUs contain only 1 fixed uncore counter.

In contrast to performance HCs, energy consumption readings through RAPL do not require this type of setup. All RAPL domains (Package, PP0, PP1, DRAM) are always active and counting since system boot. Therefore, the energy consumption can be obtained with the Raw Events’s function `read_register` that reads the MSR value. The return value from RAPL domain readings represents a raw value. Hence, it needs to be converted to Joules by applying the formula \( \frac{1}{2} \times ESU \times raw \) where ESU (Energy Status Units) is obtained with the Raw Events’s function `get_power_factor`.

### 3.3.2 Event Mux

![Diagram](image)

Figure 3.5: Top part of Figure 3.1, where it shows the relation among the KerMon components Event Mux and Raw Events, with the underlying hardware.

The **Raw Events** component provides a direct access to performance counters. Unfortunately, it also inherits its limitations, namely, the limited number of available HCs. Modern CPU architectures provide only 4 general and 3 fixed counters for each LC. In order to bypass the restricted number of HCs, a KerMon component was introduced immediately after Raw Events that provides support for monitoring more counters than it is physically available. As it can be observed by Figure 3.5 this component is designated as Event Mux. The Event Mux component
3.3 Low Level access to Hardware Counters

implements this functionality by performing time-multiplexing: swapping events at small intervals ($Cycles_{ES}$) and statistically extrapolating the counter values for the remaining intervals.

![Figure 3.6: Example of event time-multiplexing of $n$ event sets. On situation 1, the task execution is suspended, thus, Period 2 is only completed the next time that the task resumes execution.](image)

In detail, the event multiplexing functionality is accomplished by splitting the total number of events by event sets ($ES$) or groups, where each event set cannot contain more events than available HC. The number of groups ($n$) should be the minimal possible to satisfy the above condition, in order to maximize extrapolation accuracy. It also affects the multiplexing period since Period = $n \times Cycles_{ES}$, i.e., the total number of cycles where all $n$ event groups were monitored for $Cycles_{ES}$ cycles. As shown in Figure 3.6, if a context switch occurs before ending a period, when the task resumes execution, the current period also resumes from where it was, i.e., the current event group continues to count for the remaining period time. After this time is elapsed, the next event group starts counting.

$$Count(x) = \frac{Inst_{\text{Period}}}{Inst_{ES}(x)} \times Count_{ES}(x), \quad Inst_{\text{Period}} = \sum_{ES} Inst_{ES} \quad (3.1)$$

An event group contains only the counter values $Count_{ES}(x)$ for its portion of the period. The extrapolation of the final value is performed based on the number of retired instructions ($Inst_x$), that is counted by the first fixed HC. The extrapolated value $Count(x)$ is obtained by applying the Equation 3.1 In detail, the event $x$ has a counter value of $Count_{ES}(x)$, this value is then factorized with the total number of instructions in that period $Inst_{\text{Period}}$ compared with the number of instructions occurred while the event group was counting $Inst_{ES}(x)$. Thus, the final result is an extrapolation of what is expected if the event $x$ was continuously monitored during the whole period.

$$Count(x) = \frac{Inst_{\text{Period}} \times 2^{10}}{Inst_{ES}(x)} \times Count_{ES}(x), \quad Inst_{\text{Period}} = \sum_{ES} Inst_{ES} \quad (3.2)$$

Actually, in practice, Equation 3.1 is implemented as Equation 3.2 that appends the factor $2^{10}$. They are mathematically equivalent, however, in practice, make a significant difference. The main reason for this fact is that the Linux kernel does not support floating-point operations in protected mode, to avoid generating floating-point specific interruptions (e.g., divide by zero). Therefore these calculations have to be performed using unsigned 64 bits integers. This restriction causes 2 problems to Equation 3.1, namely:
3. The KerMon framework

- If $\frac{\text{Inst}_{\text{Period}}}{\text{Inst}_{\text{ES}}(x)}$ is calculated first, there will be a significant loss of accuracy, since this value will always be between 1 and the number of event groups $n$;

- Otherwise, if $\text{Inst}_{\text{Period}} \times \text{Count}_{\text{ES}}(x)$ is calculate first, it may result on integer overflow.

The solution for this problem is to first multiply $\text{Inst}_{\text{Period}}$ by a factor $(2^{10})$, then, perform the division $\frac{\text{Inst}_{\text{Period}} \times 2^{10}}{\text{Inst}_{\text{ES}}(x)}$, followed by its multiplication by $\text{Count}_{\text{ES}}(x)$. Finally, the resulting value is divided by the factor. The factor $2^{10}$ has chosen due to the following reasons:

- It is a power-of-two number, thus inexpensive shift operations can be used instead of multiplication and division, therefore minimizing the additional overhead;

- The $2^{10}$ value allows to have a rounding error smaller than $0.001$;

- It is virtually impossible to occur an overflow with this value. For example, a 5GHz LC that executes 1 instruction per cycle, requires more than 1000 hours of continuous execution (without any context switch) to cause an overflow.

Even if in-kernel floating-point operations were supported, since these operations are more expensive than integer operations, Equation 3.2 is preferable in order to reduce overhead at the expense of a small accuracy loss.

Figure 3.7 presents the usage model of the proposed Event Mux component. During the preparation stage, the function prepare_event_mux is invoked in order to distribute events into several event groups. The algorithm proposed herein aims at minimizing the number of groups, such that the extrapolation errors are also minimized. Given an unordered array of events, prepare_event_mux firstly searches the array for offcore events and distributes them. This is mainly due to the fact, that offcore events share general counters with core events, but they also impose additional restrictions: In fact, in modern Intel CPUs, only 2 offcore events can be simultaneously monitored, while it is possible to monitor 4 core events simultaneously. Therefore, distributing offcore events before distributing the core events may provide a better opportunity to form minimal number of group configurations. After offcore events are distributed, the remaining events are distributed in the same order that were provided to prepare_event_mux.

In detail, the prepare_event_mux creates and fills a data structure as depicted in Figure 3.8. This data structure contains the description of events to be monitored in each event group. The main structure is the event_mux that contains a group of fixed (core and uncore) events and an array of non-fixed events, i.e., core, offcore and uncore events. It keeps track of the total execution time (in cycles) on the total_cycles variable. Similarly, it also keeps track of the total number of retired instructions during execution on the total_metric variable.

The event_fixed_group and event_group structures have similar functionalities, and they only differ on internal attributes such as, the total number of cycles (cycles) and retired instructions (metric) occurred when the group has been physically counting events. Since is always ac-
3.3 Low Level access to Hardware Counters

Figure 3.7: Flowchart representing the usage model of the Event Mux component.

tive when the event_mux is active, the fixed group does not need to track these values, i.e., there is no multiplexing nor extrapolation on fixed events. Both event_group and event_fixed_group structures contain mapping between desired events and the real counters assigned to them, as specified in the counter_map structure. Each desired event corresponds to a counter_map instance that contains:

- The definition of the desired event, including its type and configuration;
- The real_counter index that assigns a hardware counter to this event;
- The accumulative counter value of the monitored event when the group is physically counting;
- Function pointers to the Raw Events functions. Since each event type invokes a different variant of the Raw Events functions, these pointers allow to remove conditional branches during runtime. Without conditional branches, there is no branch mispredictions, thus the expensive overheads of branch mispredictions are avoided.
3. The KerMon framework

Furthermore, the `event_mux` data is stored on successive memory locations, in order to take full advantage of cache locality. Therefore, beyond distributing events into groups, the `event_mux` data structure also optimizes the data access pattern, in order to reduce the runtime overhead.

After `prepare_event_mux` returns the `event_mux` data structure, it can be used as argument for the remaining functions. As depicted in Figure 3.7, the next function to be invoked is the `setup_event_mux`, that configures the performance counters. The HCs can be configured to start counting immediately or only after `enable_event_mux` invocation. While the task is executing, Event Mux requires a periodical invocation of the `switch_mux` function, in order to stop counting the current group, retrieve the HCs values into the corresponding `value` field of `counter_map` structure, and start counting the next event group. The `read_counter_mux` function retrieves the HCs values into the corresponding `value` field of `counter_map` structure, while the `obtain_mux_values_no_read` function uses Equation 3.2 to calculate the extrapolated counter values. When a halted task resumes execution, the multiplexing procedure resumes by invoking the `setup_event_mux` function. When a task finishes execution (exits), it is required to invoke the `clear_event_mux` function, in order to release Event Mux resources.
3.4 Event Logger

Although Event Mux can be used by any kernel-space facility, this KerMon component is designed to be invoked by the Event Logger in order to fetch monitoring samples, as shown in Figure 3.9. A sample is a data structure that stores performance and energy readings obtained in a time interval. Table 3.1 presents the information stored in each sample. Several fields in the sample structure represent relative measures (in respect to the time interval, such as the hc_values and metric fields). On the other hand, there is a set of fields with absolute values that are actually relative to the system boot information. A sample values are not further processed to avoid overheads in the task scheduler, thus they are directly obtained from the Event Mux component.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>Task’s PID</td>
</tr>
<tr>
<td>cpu</td>
<td>LC index.</td>
</tr>
<tr>
<td>ini_timestamp</td>
<td>Initial timestamp value.</td>
</tr>
<tr>
<td>end_timestamp</td>
<td>Final timestamp value.</td>
</tr>
<tr>
<td>metric</td>
<td>Retired instructions.</td>
</tr>
<tr>
<td>hc_values</td>
<td>Performance counters values.</td>
</tr>
<tr>
<td>energy.pkg_in</td>
<td>CPU Package initial energy consumption.</td>
</tr>
<tr>
<td>energy.pkg_end</td>
<td>CPU Package final energy consumption.</td>
</tr>
<tr>
<td>energy.pp0_in</td>
<td>Initial energy consumption of the CPU cores.</td>
</tr>
<tr>
<td>energy.pp0_end</td>
<td>Final energy consumption of the CPU cores.</td>
</tr>
</tbody>
</table>

Table 3.1: Data fields for the sample structure.

As it can be observed in Table 3.1, a sample contains the energy readings for different power plane domains, namely: the package (pkg) domain that measures the energy consumption of the entire CPU chip, and the PP0 (pp0) domain that measures the energy consumption of all cores and private caches. The fields in the sample structure are selected so that they provide additional flexibility. For instance, with the available sample information it is possible to process the resultant sample log to draw an application scheduling timeline.
3. The KerMon framework

The timestamp is obtained through the built-in kernel function `get_cycles`. This function induces very small amount of overheads, since it relies on the Time-Stamp Counter (TSC), i.e., a x86 register that tracks the number of cycles since boot. The timestamp can be converted to seconds by calculating \( \frac{TSC}{\text{Freq}} \), where \( \text{Freq} \) used to be the CPU clock frequency. However, since modern CPUs may dynamically change clock frequency, these CPUs use a reference TSC frequency independent of the current CPU frequency.

In addition to fetching samples, Event Logger also stores them into a buffer. This functionality is provided by relying on two specifically created sub-systems, namely Sliding Window and Ring Buffer, as presented in Figure 3.9. These sub-system have different usage scenarios: The Sliding Window is the primary sub-system that contains a small buffer designed for in-kernel consumption; while the Ring Buffer is an optional secondary sub-system, containing a bigger buffer to be consumed by user-space benchmarking tools.

3.4.1 Sliding Window

The intended purpose of Sliding Window buffer is to provide samples to the task scheduler. These samples can be used to improve scheduling decisions. Since the Sliding Window is designed to be accessed within the task scheduler, where performance is paramount, it should be a small buffer to store monitoring samples. In fact, it is allocated directly inside the task structure in order to take advantage of cache locality and thus avoid the overheads of fetching data from higher levels of the memory hierarchy. For this purpose, the Sliding Window does not require to be very big since older samples will contribute less than newer ones to the scheduling decision.

![Sample Timeline](image)

Figure 3.10: Timeline showing a sliding window with `SCHEDULERPP_SLIDING_WINDOW` = 2 slots. Initially, the Sliding Window is empty (A), then the samples 0 and 1 are stored (B) (C), the window becomes full. Thus additional samples will slide the window, first to (D) and then to (E).

The Sliding Window represents a buffer that stores the last `SCHEDULERPP_SLIDING_WINDOW` samples, where `SCHEDULERPP_SLIDING_WINDOW` is a compile-time variable (see Section 3.5 for more details). This window is sliding so that it contains the most recent samples, discarding the
3.4 Event Logger

older ones. Figure 3.10 demonstrates the sliding procedure with a window of size $\text{SCHEDULERPP_SLIDING_WINDOW} = 2$, as follows:

- The window is initially empty (A);
- When the sample 0 is taken and stored, the window holds that sample (B);
- Another sample (1) is then taken and since all samples fit in the window, no sliding occurs (C);
- At this stage, the window is already full. Hence, when the sample 2 is introduced, it does not fit on the window. Therefore it slides so that the new sample 2 is inserted in the last position of the window and the older sample, i.e. the sample in the first position (0), is discarded (D).
- From now on, the newer samples will oblige the window to slide in order to fit and discard the older ones.

![Figure 3.11: Implementation of the Sliding Window with SCHEDULERPP_SLIDING_WINDOW = 8 slots. The cursor points to the slot to be written and it loops when it reaches the end of the buffer.](image)

While Figure 3.10 presented the Sliding Window in the conceptional terms, the practical implementation of Sliding Window is depicted in Figure 3.11. There are two defined variables, namely: `count` for the cursor index, and `full_window` that flags if the window is fulfilled. Initially, both variables are set to 0 (false). After a sample is stored, `count` is incremented. When `count` is equal to `CONFIG_SCHEDULERPP_SLIDING_WINDOW`, it is reset to 0 and the `full_window` flag is set to 1 (true). When this occurs, the sliding window starts to overwrite the oldest sample that is pointed by the `count` variable.

The Sliding Window functionality is deeply incorporated into the OS task scheduler. For this, it provides `task_restart_execution`, `task_get_samples` and `task_hold_execution` functions that read the performance counters, TSC and energy consumption values. All of these functions rely on the previously described Event Mux functionality.

Figure 3.12 represents the proposed Scheduler++ algorithm (see Section 3.5) while a task is executing, as well as task execution stages where Sliding Window functions are invoked (see gray rectangles). When the OS task scheduler selects a Scheduler++ task to be executed (3), the `task_restart_execution` function is invoked (A), i.e., the start of a task execution session. This function obtains the values for the sample’s ini fields (see Table 3.1), which correspond to the TSC and RAPL interfaces. Since these counter values are relative to boot time, it is necessary
3. The KerMon framework

Figure 3.12: Detail of Figure 3.17: Task execution flow on Scheduler++. The gray rectangles depict task scheduler modifications, in order to invoke Sliding Window functionality.

to have the initial values to compare the number of cycles and energy consumed on the current sample. The task_restart_execution also invokes the setup_event_mux function of the Event Mux component, in order to configure HCs.

While a task is running (4), periodically occurs a scheduler tick that updates statistics (5). If the event multiplexing interval has elapsed (B), the switch_mux function of the Event Mux component is invoked (C). This function stops the current event group from counting in order to allow the next group start/resume counting. If the number of cycles past the last sample is equal or bigger than the configured sampling interval (D), a sample is taken by invoking the task_get_samples function (E). The task_get_samples is designed to be invoked on a scheduler tick, to obtaining middle-of-session samples, i.e., monitoring samples that do not match the beginning or the end of an execution session. The middle-of-session samples allow a more precise and uniform monitoring granularity between samples, which is specially useful for benchmarking. The task_get_samples completes the current sample by filling all fields with the values provided by the obtain_mux_values__no_read function of Event Mux component (except for the ini fields that are already filled). Then, the sample is introduced in the Sliding Window buffer. Finally, task_get_samples creates a new sample by filling the ini fields.
3.4 Event Logger

When a context switch occurs, i.e., the execution session ends, the task_hold_execution function is invoked. The task_hold_execution invokes the Event Mux functions read_event_mux and obtain_mux_values_no_read in order to complete the current sample by filling all fields (except for the ini fields that are already filled). Then, the sample is introduced in the Sliding Window buffer.

All samples from the same session have the same initial values, thus all samples range from the beginning of the execution session until the moment these are taken/completed. This allows to optimize the execution of the proposed monitoring mechanism by avoiding to reset the counters every time that a middle-of-session sample is taken, thus allowing to avoid unnecessary overhead in the task scheduler, where performance is paramount. When analyzing benchmark logs, the values for a specific sample can be obtained by subtracting its values from the values of the previous sample (of the same session). This feature is actually offsetting kernel-space overhead to the analyzer. Implicitly, this also allows to identify the sessions of a task by looking into the ini_timestamp field on the benchmark log, since all samples of a session will have the same ini_timestamp value.

3.4.2 Ring Buffer

As mentioned before, the Sliding Window must be small in order to reduce overall overhead related to memory access. Although a small buffer size is enough for kernel purposes, it is too small for benchmarking purposes, where the data needs to be transferred to the user-space memory in order be further analyzed. This information can be used, for example, to create a CARM or ORM plot, in order to detect performance bottlenecks. Therefore, the proposed Event Logger introduces an optional functionality, a sub-system that provides monitoring samples to user-space applications, designated as Ring Buffer. The Ring Buffer contains a much bigger buffer than Sliding Window’s buffer in order to allow a reasonable time interval for a user-space benchmarking tool to fetch samples with a low probability of losing samples due to a full buffer. The Ring Buffer is implemented as a circular buffer, where is commonly known as ring buffer in Linux kernel terminology. The Linux kernel provides some helper functions to manage ring buffers, but it does not provide a complete generic implementation. In fact, Linux relies on ring buffers to implement some of the core facilities, such as the syslog logging subsystem.

The Ring buffer (or circular buffer) is a specific type of buffer: implemented as a fixed-size array which end is connected to the beginning. This design allows the removal of elements without any data shifting, thus all operations on a ring buffer has a constant and small overhead. Ring Buffer also defines 2 actors that access it, namely:

- The **Producer** that inserts data into the buffer. In KerMon, the **Producer** is the Sliding Window. Whenever the Sliding Window cursor loops to the beginning, it inserts all samples of its window to the Ring Buffer;
3. The KerMon framework

- The **Consumer** that reads and removes data from the buffer. The intended Consumer is a user-space benchmarking application that can transfer the buffer data through the specifically designed system call `schedulerpp_read_event_log`. In KerMon, the **Consumer** is the `kernelCount` using the `libSched++` library (see Section 3.6).

![Ring Buffer diagram](image)

**Figure 3.13:** Example of a ring buffer with 8 slots. It has 2 cursors: The `head` points to the slot to be written while the `tail` points to first non-consumed data. When anyone points to the slot 7 and it is incremented, it will loop back to point to the slot 0.

The **Ring Buffer** functionality is defined according to the following set of rules:

- A ring buffer has 2 cursors as shown in Figure 3.13:
  - `head`, to designate the slot where the **Producer** writes the sample;
  - And the `tail` cursor which points to the sample that is read and then removed by the **Consumer**;

- When the `tail` and `head` point to the same element means that the buffer is empty;
- When an element is inserted, the `head` is incremented;
- After reading an element, the `tail` is incremented;
- To remove the oldest element, it is just required to increment the `tail` cursor (therefore, when an element is read, it is automatically removed from the buffer);
- When the buffer is full, the `head` cursor points to the slot immediately behind of the `tail`;
- When any cursor reaches the end of the buffer, it will loop back to the beginning.

Figure 3.13 shows an example of the **Ring Buffer** with 8 elements. In the initial state, both `head` and `tail` cursors point to element 0. Figure 3.13 represents the buffer state when 4 elements are inserted in the buffer by the **Producer**. For each produced element, the `head` cursor jumps to the next slot. When an element is consumed, the `tail` is also incremented, thus effectively removing it from the buffer. When any cursor tries to reach the position 8, it actually wraps into position 0.

The Linux kernel provides two useful features that help in the implementation of the circular buffers. Firstly, there is a set of macros to obtain information about $2^n$ sized ring buffers such as occupancy. On a generic circular buffer, to implement functions that access the buffer requires
the usage of the slow modulus instruction or through a conditional branch. However, by restricting the buffer size to the $2^n$ number, it is possible to use the faster bitwise AND instruction instead of the modulus operand. The Linux kernel also provides the memory barrier functionality. The compiler and/or CPU may execute instructions in a different order than the one that it is programmed, although it always respects the dependencies between instructions. A memory barrier is an instruction that imposes a partial ordering over the memory operations [33]. In this implementation of ring buffer, memory barriers guarantee that the data is read (by the Consumer) or inserted (by the Producer) before the Tail or Head is updated, respectively. Therefore, it is not required to use a locking synchronization mechanism between the Producer and the Consumer [34].

By default, each task has its own ring buffer. However, KerMon allows benchmarking tools to configure a target application to share the ring buffer among its children (threads or processes). This feature allows benchmarking tools to fully analyze an application by accessing only one ring buffer. In this case, there is multiple Producers, thus a synchronization mechanism is necessary. The Producers execute on the task scheduler context, where sleeping functions can not be invoked. Therefore, the only classical synchronization locking mechanism available is the busy-wait lock, also known as spinlock, that actively loops itself until the lock is available. In contrast, since Consumers are regular user-space application, a (sleeping) mutex is applied to avoid simultaneous access to the same buffer by multiple Consumers.

![Figure 3.14: Usage model of Ring Buffer functions.](image)

Figure 3.14 show the available Ring Buffer functions and the scenarios for their invocation. When a task firstly executes on Scheduler++, the scheduler invokes the function `create_raw_log` to initiate the ring buffer functionality. In the scenario where a task is forked from a Scheduler++ task sharing its ring buffer, the child task must invoke the `register_me_in_raw_log` instead. This
function simply notifies that there is one more Producer for that ring buffer.

The Sliding Window writes into the corresponding Ring Buffer when its window is completely filled with new samples, by invoking the function \texttt{write\_raw\_log} that copies the window content into the ring buffer. Then, this data can be retrieved by user-space applications through the system call \texttt{schedulerpp\_read\_event\_log}.

Finally, when a task exits (or changes class from Scheduler++), the function \texttt{unregister\_me\_in\_raw\_log} must be invoked in order notify Ring Buffer that there is one less Producer. Next, if the ring buffer does not have any Producer, its resources are freed.

\[ nPages = \frac{size - 1}{PAGE\_SIZE}, \quad size = f(SCHEDULERPP\_LOGGER\_SIZE, nEvents) \]  \hspace{1cm} (3.3)

\[ order = \begin{cases} \log_2(nPages) + 1, & \text{if } nPages > 0 \\ 0, & \text{otherwise} \end{cases} \] \hspace{1cm} (3.4)

Usually, memory allocations are performed by calling the \texttt{kmalloc} function. However, this function is not recommended to allocate more than a few kilobytes \[35\]. To allocate more memory than \texttt{kmalloc}, the alternative function \texttt{__get\_free\_pages} is used herein to obtain complete control over the memory pages. Since \texttt{__get\_free\_pages} is a lower level function than \texttt{kmalloc}, it provokes less performance overheads at the cost of reduced memory granularity. On x86 systems, a regular memory page has 4096 bytes \((PAGE\_SIZE)\), and it represents the minimal amount of memory that can be allocated. Furthermore, \texttt{__get\_free\_pages} only allows to allocate \(2^{order}\) number of pages, since it accepts the \texttt{order} argument. In order to determine the number of pages, Event Logger relies on Equations 3.3 and 3.4. Equation 3.3 firstly determines the memory size of a ring buffer using the \texttt{SCHEDULERPP\_LOGGER\_SIZE} compile-time definition and the number of events to monitor \texttt{nEvents}. Then the \texttt{nPages} value is calculated, where \(nPages + 1\) is the number of necessary pages to fit the ring buffer, rounded-up to the next integer. After the \texttt{nPages} is determined, it is necessary to obtain the rounded-up \texttt{order} that complies with the \(2^{order}\) requirement of \texttt{__get\_free\_pages}, by using the Equation 3.4.

### 3.5 KerMon scheduling classes: Scheduler++ and CFS++

As referred on Section 2.3, Linux implements several scheduling classes that implicitly define the order of priority among the tasks. As shown in Figure 3.15, there are 3 main scheduling classes, namely: Deadline, Real-Time and Fair (see white rectangles). KerMon implements two additional classes, shown as the gray rectangles in Figure 3.15: Scheduler++ and CFS++. The new scheduling classes are inserted such that the general algorithm of the task scheduler to pick a new task (i.e. the priority order among classes) is depicted as in Figure 3.15. Therefore, the task scheduler picks a task to execute, by querying on each scheduling class in a predefined
3.5 KerMon scheduling classes: Scheduler++ and CFS++

order, until one of them returns a task. The order is defined as follow: Deadline, Real-Time, Scheduler++, CFS++ and lastly, the default Fair scheduling class. As before, it is implicitly assumed that a former class have a higher priority than the latter ones. Therefore, the KerMon classes have higher priority than the default one that is the Fair scheduling class.

Attributing higher priority to a benchmark that executes in Scheduler++ (than to remaining system applications), provides execution isolation. This means that the remaining applications, which run on Fair scheduling class, have insignificant effect on the Scheduler++ benchmarks. Thus, this isolation feature contributes significantly to an accurate application monitoring and characterization, namely, to identify application/system bottlenecks through CARM and ORM. On the other hand, the benchmarks that are executed in the regular Fair class do not benefit from this isolation, thus other applications may interfere with benchmark results. The CFS++ is a clone of Fair class where only slight modifications were introduced to allow its co-existence with the original class. Since this Fair clone also executes on a higher priority, it also takes advantage of implicit isolation and it allows performing a fair comparison between the Fair scheduling class and the Scheduler++ class in terms of the induced overhead of KerMon.

The Scheduler++ is a modified Fair scheduling class that invokes the kernel monitoring facility Event Logger, as depicted in Figure 3.16. When a scheduler tick occurs, Scheduler++ might invoke Event Logger functionality in order to take a sample or to switch the current multiplexing event group (A). When a context switch occurs, the Event Logger functionality must be invoked take a sample and/or configure HCs (B).

Figure 3.17 presents the comparison of Fair and Scheduler++ classes. The white boxes belong to both classes, while the gray boxes belong only to the Scheduler++ scheduling class. 
3. The KerMon framework

Figure 3.16: Section of Figure 3.1 where it shows the invocation of Event Logger functionality by the Scheduler++.

Figure 3.17: Task Lifecycle on Scheduler++. This figure includes the gray boxes that depict invocation of Sliding Window functions.

The details behind the white boxes were already described in Section 2.3.3 when explains the flow of the CFS algorithm. The additional gray boxes (invocation of Kernel Logger) is explained in Section 3.4.1. In brief, the functionality of the Scheduler++ can be summarized as follows:

- When a task is created (1), it becomes runnable and it is inserted on the runqueue (2);
- When Scheduler++ selects this task (3), it setups the event counters and takes the initial values (A);
- While the task is running (4), the scheduler tick is periodically executed to update the following statistics (5):
  - If the task’s virtual runtime (6) is less or equal than the ideal runtime, the task continues its execution. In this case:
3.5 KerMon scheduling classes: Scheduler++ and CFS++

1. **Scheduler++** checks if the event multiplexing interval has expired (B). If that is the case, the currently counting group stops counting in order to allow the next group start/resume counting (C);

2. Then, **Scheduler++** checks if the sampling interval has expired (D). If that is the case, then it creates a new sample, completing the older one (E);

3. The task resumes execution (4);

   - Otherwise, if the task’s virtual runtime (6) is bigger than the ideal runtime, the task needs to be preempted (7);

• When the task stops running, the current sample is completed (F). Hence, the task changes its state to a halted state (8). Depending on the context switch reason (9), the task will take the following actions:

   - The task will be directly inserted back to the runqueue (2), if it was preempted;
   - The task sleeps (10), while waiting for an event. When the event occurs (11), the task is inserted back in the runqueue (2);
   - The task exits, ending its lifecycle (12).

---

Figure 3.18: Example of a timeline of two tasks sharing a LC, where task A executes on **Scheduler++** while B executes on **Fair** scheduling class. In this example, the sampling period is 50ms and the event multiplexing switching is disabled.

Figure 3.18 presents an example timeline of two tasks executing on the same logical processor, where task A executes on Scheduler++ scheduling class, while task B executes on the
default Fair scheduling class. The task $B$ is executed first (1), however, as soon as that task $A$ becomes runnable (2), the task $B$ is preempted, so that the LC executes task $A$. As it can be observed, applications scheduled on the Scheduler++ class have higher priority than those on the Fair class. Thus, task $B$ will not execute while task $A$ is runnable. Whenever the task $A$ is about to execute, the performance counters are configured and start counting. On situation 3, task $A$ sleeps, waiting for a resource. Whenever task $A$ releases the LC the performance counters are read into a sample. Since the LC is free, the task $B$ can resume execution. When task $A$ awakes (the resource is now available) on situation 4, task $B$ is preempted and the performance counter are configured just before task $A$ resumes execution. While task $A$ is executing, periodically occurs scheduler ticks. Usually, the scheduler tick only updates statistics (situation 5 and 7). However, if the sampling interval has elapsed (6), a new sample is taken. When task $A$ finishes execution (8), one last sample is taken. Then, task $B$ can resume execution.

![Usage model for the KerMon scheduling classes.](image)

An application is automatically monitored when its scheduling class is set to Scheduler++. This is performed by assigning the new scheduling policy SCHED_PP to the application. This can be performed through the already existing system call sched_setscheduler, as shown in Figure 3.19. Similarly, a user can set an application to run on the CFS++ scheduling class by assigning the policy SCHED_CFSPP to the application. Users can use the standard chrt command line tool, that invokes sched_setscheduler, in order to change an application’s scheduling policy and therefore its scheduling class. In addition, Scheduler++ also implements a new set of system calls to allow users to select monitoring events, namely:

- The schedulerpp_set_profile system call defines a profile, i.e., selection of events, sampling interval and event multiplexing switching interval. Hence, standard profiles controls simultaneously the monitoring events used by several tasks, since it has a system-wide scope. The standard default profile is the default event configuration for Scheduler++ tasks, thus it should be configured on system boot;

- A Scheduler++ task can ignore standard profiles if a user sets a custom profile by invoking
3.6 User-Space tools

the schedulerpp\_set\_events system call that defines task-specific events, sampling interval and event multiplexing switching interval. All children created afterward will inherit the same properties;

- On the other hand, the system call schedulerpp\_reset\_events will change a task's profile to the standard default one;

- The schedulerpp\_execute system call combines the sched\_setscheduler and schedulerpp\_set\_profile functionality in only one call;

- The schedulerpp\_get\_info system call allows user-space applications to obtain information such as the CPU clock frequency, useful to determine time intervals.

The Scheduler++ uses the same runtime variables used in the Fair scheduling class, thus any change on these variables will affect both classes identically. However, the functionality of Scheduler++ can be configured via compile-time definitions. These rely on the Linux kernel Kconfig facility that allows a user to easily customize the kernel in order to specialize the kernel to one's needs.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEDULERPP</td>
<td>No</td>
<td>Activates KerMon functionality.</td>
</tr>
<tr>
<td>SCHEDULERPP_SLIDING_WINDOW</td>
<td>5</td>
<td>Sliding Window buffer size.</td>
</tr>
<tr>
<td>SCHEDULERPP_LOGGER_SIZE</td>
<td>1024</td>
<td>Ring Buffer size.</td>
</tr>
<tr>
<td>SCHEDULER_CFSPP</td>
<td>No</td>
<td>Activates the CFS++ scheduling class.</td>
</tr>
</tbody>
</table>

Table 3.2: Main compile-time definitions present under the Kconfig menu entry Scheduler++

To easily tweak KerMon behavior, Table 3.2 lists the major definitions introduced by KerMon. The SCHEDULERPP definition is designed such that if it is inactive (set to No), the compiled kernel should be the same as the unmodified Linux kernel. The SCHEDULERPP\_SLIDING\_WINDOW defines the number of samples that the Sliding Window buffer can hold. Similarly, the SCHEDULERPP\_LOGGER\_SIZE defines the number of samples that the Ring Buffer can hold. However, for performance reasons (see Section 3.4.2), this value must be a power-of-two number, e.g., 1024. The Ring Buffer is a optional feature, that can be disabled by setting SCHEDULERPP\_LOGGER\_SIZE to 0. This options allows to completely remove Ring Buffer overheads (and functionality) if there is no intention of retrieving the event monitoring values from the kernel. It is possible to observe, by the SCHEDULER\_CFSPP definition, that the CFS++ scheduling class is also an optional feature, since it is mainly used to compare Scheduler++ with the Fair scheduling class, as already mentioned.

3.6 User-Space tools

Although KerMon focuses mainly on kernel-space by providing a performance and energy monitoring framework that can be invoked by kernel-space instructions. It is also a viable and
3. The KerMon framework

Figure 3.20: Section of Figure 3.1 where it shows KerMon user-space tools: libSched++ and kernelCount.

A simple solution for user-space benchmarking purposes. Therefore, it provides several user-space tools specially designed for this purpose, namely libSched++ and kernelCount, as presented in Figure 3.20.

3.6.1 libSched++

The libSched++ library provided by KerMon allows developers to implement custom-made benchmarking tools. It is a collection of useful function modules put together in one package namely:

- Functions that access directly the CPU information (via the CPU's CPUID interface);

- It allows to write Gnuplot-friendly data files and automatically generate scripts to graphically present the benchmarking results, namely to create CARM and ORM plots. A single CARM plot models the performance upper-bounds of multi-core architectures having into account the different memory levels. Thus, this libSched++ functionality provides support to perform an accurate characterization of a target application on a modern multi-core system in order to identify bottlenecks as described on Section 2.4.

- The libSched++ identifies and normalizes the middle-of-session samples, i.e., obtains the counter values specific for the current sample, by subtracting its values from the values of the previous sample (of the same session), as described on Section 3.4.1. It also calculates derived values such as:
  - Power \( (Power = \frac{Energy}{Time}) \);
  - Task time \( (Task\_cycles = \sum Sample\_cycles) \), that is the effective time that the task was executing on a CPU.

- It provides direct wrappers to the KerMon system calls as shown in Figure 3.20.
3.6 User-Space tools

- It supports some cross-platform functionality through preset events. This is a functionality similar to the [PAPI]'s preset events functionality (see Section 2.1.2).

![Preset Event Composition Diagram]

Figure 3.21: Composed preset event example, where the preset event is composed by operations among multiple raw events. The operations are performed in the left-to-right order.

In detail, libSched++ provides support for preset events, in order to hide the real raw events from the client application, providing some cross-platform abilities. Moreover, these preset events can be composed, i.e., one preset event can be mapped to several raw events, where each one is related to each other by a mathematical operation. Figure 3.21 shows an example of a composed Preset Event. This Preset Event is composed by 4 Raw Events (0 to 3) and 4 operations between them. When libSched++ configures an application monitoring events, it sends to Scheduler++ the Raw Events. When retrieving the counter value for each Raw Event (Value), libSched++ will perform the operations on the values, in a left-to-right order, and outputs the resulting value. Therefore, in this example, the resulting Value of the preset event is Value = (Value_0 + Value_1) * Value_2 + Value_3. This functionality does not hinder flexibility, since libSched++ also accepts raw events.

A client application defines a standard profile, through the function set_profile, or a custom profile, through the function set_events. It is also possible to set events (custom profile) while changing the target application to execute on Scheduler++ by using the function set_scheduledpp.

3.6.2 kernelCount

The second user-space tool that KerMon provides is the kernelCount, which allows to monitor an application by using the KerMon facilities. Thus, a non-developer user can immediately perform benchmarking with the provided KerMon tools. Figure 3.20 shows that kernelCount does not invoke the KerMon kernel-space functionalities, instead it uses libSched++ for that purpose, therefore it takes advantage of the additional features.

kernelCount is a console application that accepts arguments. The first argument must be a comma-separated list of events to monitor. Then, the remaining arguments compose the command of the target application. For instance, if the target application is usually executed by typing the command "targetApp arg1 arg2 arg3", using kernelCount, the final command must be "kernelCount event1,event2,event3,event4 targetApp arg1 arg2 arg3".

The first step for kernelCount, as can be observed in Figure 3.22, is to process the arguments, mostly to fill the libSched++'s event elements. Then kernelCount forks, creating a child...
process. The child process setups the performance monitoring by invoking the libSched++’s function set_schedulerpp, which changes the child’s scheduling class to the Scheduler++ and sets the events to monitor. After this, the child process invokes the exec system call that replaces its process image with the target application, thus the child becomes effectively the target application. Therefore, the child process dies when the target application dies.

As Figure 3.22 depicts, while the child is running, the parent process performs the following loop:

1. It sleeps until a predefined time interval elapses (0.5 seconds);
2. When awakes, if the child died, kernelCount exits. Otherwise, it invokes the Ring Buffer’s read_event_log system call in order to retrieve samples;
3. Then, it stores the retrieved samples in a Gnuplot-friendly tab-separated file;
4. It loops back to sleep (step 1).

3.7 Summary

In this chapter, it was described in detail the work developed for this thesis, that is designated as KerMon, with the goal of providing a performance and energy monitoring framework that can
3.7 Summary

be used in kernel-space. Furthermore, it must be capable of executing on the task scheduler con-
text, in order to be truly universal. This context executes with disabled interrupts and therefore has
restrictions such as not being able to invoke functions that might sleep. Moreover, it is executed
every few milliseconds, further imposing more restrictions in order to avoid hindering the overall
system performance. Therefore, any KerMon code that executes in the task scheduler context
must consider performance as the paramount value.

KerMon is composed by several components, mostly in kernel-space:

- The Raw Events is the component responsible for obtaining performance and energy read-
ings;
- The Event Mux implements time multiplexing in order to measure more events than what
is physically possible;
- The Event Logger aggregates these readings into samples and stores them in a buffer;
- The Scheduler++ scheduling class controls the overall KerMon mechanism.

KerMon, beyond being a framework, it already implements an actual benchmarking application.
The tools that it provides are the following:

- The libSched++ library that allows user-space developers to fully benefit of KerMon facil-
ities;
- And the kernelCount application that allows non-developers to benchmark applications.

The data obtained through KerMon can be analyzed in order to identify the bottlenecks that
hinders the overall performance. For that purpose, KerMon provides facilities in order to generate
performance models, namely the CARM.
3. The KerMon framework
# Experimental Results

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Experimental Environment</td>
<td>58</td>
</tr>
<tr>
<td>4.2 Overhead Analysis</td>
<td>60</td>
</tr>
<tr>
<td>4.3 Performance Monitoring</td>
<td>63</td>
</tr>
<tr>
<td>4.4 Energy Monitoring</td>
<td>71</td>
</tr>
<tr>
<td>4.5 Summary</td>
<td>72</td>
</tr>
</tbody>
</table>
4. Experimental Results

In order to evaluate the KerMon framework for in-kernel performance and energy monitoring, an extensive experimental evaluation was conducted by relying on the standard SPEC CPU2006 benchmark suite. This chapter presents the evaluation of KerMon. Thus, it begins by describing the experimental environment. Then, experimental tests are performed using this environment to assess the KerMon overhead impact on the overall system performance and to characterize several SPEC CPU2006 benchmarks by using CARM plots as discussed on Section 2.4.

4.1 Experimental Environment

Table 4.1: Benchmarking environment configuration.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7 3770K @ 3.5GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>Corsair Vengeance DDR3 2x4GB @ 1.6GHz</td>
</tr>
<tr>
<td>MotherBoard</td>
<td>ASUS P8Z77-V LX</td>
</tr>
<tr>
<td>OS</td>
<td>openSUSE 13.1 (Bottle) (x86_64)</td>
</tr>
<tr>
<td>Kernel</td>
<td>3.14.3-antaonux-RC1</td>
</tr>
<tr>
<td>Benchmark Suite</td>
<td>SPEC CPU2006</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.8.1</td>
</tr>
<tr>
<td>SPEC Compiler</td>
<td>Intel Parallel Studio XE 2013</td>
</tr>
</tbody>
</table>

Table 4.1 briefly describes the benchmarking environment. For experimental purposes, the benchmarks are executed on Adriana machine, an Intel platform system with a quad-core Intel Core i7 3770K [CPU] (Ivy Bridge family), operating at a nominal clock frequency of 3.5GHz (although it can reach as high as 3.9GHz due to the Turbo Boost technology [36]). While this [CPU] is composed by 4 physical cores, it is capable of executing 8 simultaneous threads (it has 8 [LCs]), since it supports Hyper-threading. This [CPU] supports hardware performance monitoring, where each [LC] contains 4 general [HCs] and 3 fixed [HC]. It also supports RAPL’s Package and PP0 domains for energy monitoring (See Section 2.1).

The system memory hierarchy is the following, from the lowest levels to the top:

- There are two [L1 caches] in each physical processor core, one for data and other for instructions. These caches are shared among the [LCs] of a core. Each [L1 cache] is capable of storing 32kB;
- There is one unified [L2 cache] for data and instructions per physical core, which it can hold 256kB;
- The [L3 cache] is located in the [CPU]‘s uncore domain and it is shared among all (logical) cores. It is the biggest in-chip cache, capable of hosting 8192kB;
- The [DRAM] is the main system memory and is capable of storing 8GB. It is composed by 2 memory banks, each storing up to 4GB. They operate at 933MHz, in dual-channel configuration, thus doubling the data bandwidth.
### 4.1 Experimental Environment

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEDULERPP</td>
<td>Yes</td>
<td>Activates KerMon functionality.</td>
</tr>
<tr>
<td>SCHEDULERPP_SLIDING_WINDOW</td>
<td>5</td>
<td>Sliding Window buffer size.</td>
</tr>
<tr>
<td>SCHEDULERPP_LOGGER_SIZE</td>
<td>1024</td>
<td>Ring Buffer size.</td>
</tr>
<tr>
<td>SCHEDULER_CFSPP</td>
<td>Yes</td>
<td>Activates the CFS++ scheduling class.</td>
</tr>
</tbody>
</table>

Table 4.2: Main compile-time definitions present under the Kconfig menu entry Scheduler++.

The system runs a 64-bits Linux OS, namely OpenSuse 13.1. The operating Linux kernel is a modified 3.14.3 kernel in order to include the KerMon framework. The kernel compile-time configuration is the default one for this OS, except for the new definitions introduced by KerMon, which are defined in Table 4.2 (see Section 3.5). In brief, the Scheduler++ and CFS++ scheduling classes are active, the Sliding Window buffer contains the last 5 samples and the Ring Buffer can store 1024 samples.

The custom kernel and the KerMon user-space tools are compiled using GCC 4.8.1, while the benchmark applications are compiled using an Intel compiler toolchain, namely, the Intel Parallel Studio XE 2013. The benchmark applications are provided by the SPEC CPU2006 suite [37], which is an industry-standardized, CPU-intensive benchmark suite, stressing a system’s processor and memory subsystem capabilities. For the experimental tests, the floating-point benchmarks are executed using KerMon and the resulting data is analyzed.

Before executing a benchmark, the following steps are performed:

1. The CPU clock frequency is fixed to 3.5GHz. This is performed by writing userspace into the file `/sys/devices/system/cpu/cpu$i/cpufreq/scaling_governor` and `3500000` into `/sys/devices/system/cpu/cpu$i/cpufreq/scaling_setspeed`, where $i$ is the LC index. In fact, the LC frequency can be varied in a range between 1.6GHz and 3.9GHz. However, the LCs clock frequency was fixed in order to guarantee a more consistent experimental evaluation;

2. The Linux kernel has a functionality designated as Non Maskable Interrupt (NMI) Watchdog that detects hard lock-ups. This functionality takes advantage of the last general HC, thus it generally conflicts when accessing the performance counters. Since KerMon might also use this HC, the NMI Watchdog was disabled, by writing 0 into the file `/proc/sys/kernel/nmi_watchdog`.

The benchmarks are executed by invoking a specially custom-made script that executes kernelCount (See Section 3.6.2) with the commands from the `speccmds.cmd` file under each benchmark run directory. In case that it contains multiple commands, each command generates a different benchmark data file with a suffix `._i`, where $i$ is the index of the command (starting by 1). Unless otherwise stated, on every benchmark experimental evaluation, the benchmark application is the only application actively executing on the system, except for system background...
processes and kernelCount. It is worth empathizing that the background processes have insignificant interference, especially since benchmarks are executed on a scheduling class with higher priority than the background processes. The kernelCount application, after creating a child process running the benchmark application, continues to execute on the lower-priority regular Fair scheduling class in order to fetch benchmark data from KerMon every 0.5 seconds. Unless otherwise stated, the CPU affinity of the benchmark applications was set on the first LC (0). The purpose of this imposition is to replicate the execution conditions on every benchmark, otherwise, the tasks may execute on different LCs or the task scheduler might migrate them in the middle of execution.

4.2 Overhead Analysis

During the development of KerMon, a special attention was payed to the induced overheads and the performance impacts on the overall system level, since KerMon is integrated within the OS task scheduler. In order to assess the KerMon impact on the overall performance, an experimental evaluation based on the total execution time comparison is performed. In this experiment, SPEC CPU2006 benchmarks are executed on CFS++ and Scheduler++ (See Section 3.5). The CFS++ scheduling class is functionally identical to the default Fair scheduling class, except for the fact that tasks executing on it have higher priority than tasks executing on the Fair scheduling class. This fact allows to perform a fair comparison between Fair and the Scheduler++ scheduling classes.

Figure 4.1 shows the difference of the total execution time between a benchmark executed on CFS++ and Scheduler++, where the total execution time is the average time of 10 benchmark executions. The Scheduler++ is configured to count 8 events divided in 2 groups, with a multiplexing switching interval of 50ms and a sampling interval of 100ms. The difference is expressed in percentage according to the calculation of \(100 \times \frac{Time_{Scheduler++} - Time_{CFS++}}{Time_{Scheduler++}}\), where \(Time_s\) is the total execution time on the s scheduling class. If the resulting value is positive, the benchmark execution on Scheduler++ has a bigger execution time than on CFS++, otherwise, if it is negative, the benchmark is faster on Scheduler++.

Figure 4.1 shows that, in some cases, the task executing on Scheduler++ appears to be actually faster than executing on the CFS++ scheduling class (e.g. for milc, gromacs, calculix, tonto, lbm and sphinx3). However, it is worth to emphasize that it is very difficult to assess the correct runtime overhead value in kernel-space since it depends on many factors (such as fluctuation of the pressure on the memory subsystem, system changes, execution uncertainties and other non-deterministic behaviors). In this experimental evaluation, the average difference of all benchmarks is 0.27%. However, this average may not be completely regarded as the overhead of KerMon, since execution time differences are very low. In fact, these results may rather serve
4.2 Overhead Analysis

![Graph showing execution time difference between Scheduler++ and CFS++](image).

Figure 4.1: Total execution time difference (%) between Scheduler++ and CFS++.

as a hint that the KerMon overheads are insignificant.

In order to further assess the overheads introduced into the kernel, the functions that are frequently invoked were tested in user-space. Beforehand, an unit-testing facility was developed for debugging KerMon on a controlled environment in user-space. Using this facility, an experimental test was performed by invoking each function $10^9$ times. The elapsed time is then divided by the number of iterations ($10^9$) in order to obtain the average time of each iteration. The evaluated functions are the following:

- the `switch_mux` function that switches the currently counting multiplexing event group in order to allow other groups to have their share of counting time. It is invoked on every multiplexing switching time interval;

- the `task_get_samples` function that retrieves the raw values, extrapolates and compiles them into a sample. Then, it stores the sample into an Event Logger buffer. It is invoked on every sampling time interval;

- the InOut is a pair of functions (`task_restart_execution` and `task_hold_execution`), that are invoked when execution session starts and ends, respectively. In other words, they are executed on context switches. Since these functions work in tandem, they are evaluated together.

The results obtained when evaluating the above referred functions are presented on Figure 4.2. The testing methodology also considered the optimization level applied during the compilation.
of Linux Kernel (O2) and the O0 level, i.e., no optimizations. In brief, as it can be seen, the
\texttt{task\_get\_samples} is the function that requires more time to execute. By default, the Linux kernel
compiles using the optimization level 02, thus, initially, this was the optimization level used for
this overhead benchmarking. When using this compiler optimization level, the \texttt{task\_get\_samples}
function required about 10$\text{ns}$ to execute (i.e., 27$\text{ns}$ less than with O0 level). Effectively, this very low
time was probably achieved due to optimizations performed by compiling in this unit-test scenario
(which are not possible to perform in the real kernel scenario). For instance, the loop that executes
each function 10$^9$ time may be optimized through loop unrolling, while the kernel does not invoke
these functions in a sequential loop. Therefore, to make a fair overhead assessment, the compiler
optimizations are disabled by defining the O0 optimization level on this overhead benchmarking.

\texttt{KerMon} is designed to take advantage of cache locality in order to reduce cache misses,
thus reducing the corresponding overheads. In the unit-test scenario, since functions are exe-
cuted in a sequential loop, it is expected a reduced number of cache misses. In the real-world
kernel scenario, it is unpredictable the number of cache misses when the OS and applications
are competing for the memory resources, although it is expected that the impact of cache misses
in the real-world scenario is bigger than in the unit-test scenario. Therefore, the results of this
analysis do not consider cache misses caused by resource contention.

Figure 4.2 has the invocation timing of the most invoked functions. These timings were ob-
tained using the Kernel's compiler optimization level (O2) and with no optimization (O0). It is pos-
sible to notice that the compiler could not further optimize the `switch_mux` function. This function has an overhead near 3\(\text{ns}\). Therefore, the KerMon multiplexing facility features a practically nonexistent overhead during runtime.

As previously referred, `task_get_samples` is the function that adds more overhead. In the worst case scenario, i.e., when optimizations are disabled, it adds almost 37\(\text{ns}\). Assuming a sampling interval of 100\(\text{ms}\), it will add 0.37\(\text{ms}\) per second of execution, i.e., an overhead of 0.037% of the total execution time. In the kernel scenario, that is compiled using the optimization level 02, it will add an overhead of 0.010% of the total execution time. As expected, this overhead is insignificant, even when the function is not optimized by the compiler. The runtime overhead on the kernel should be near 10\(\text{ns}\) per invocation.

The overheads introduced on a context switch, in order to start and/or finish the execution session, are also shown in Figure 4.2 in the InOut bars. The non-optimized functions reach near 21\(\text{ns}\) of execution time per session. Hypothetically, if sessions had a 100\(\text{ms}\) period, the increased overhead of the non-optimized functions will be 0.021% of the total execution time, while the optimized functions will only add 0.010%. As it can be observed, this overhead is also insignificant, even when the functions are not optimized by the compiler. The runtime overhead on the kernel should be near 10\(\text{ns}\) per execution session.

In brief, the execution scenario considered herein assumes run-time monitoring of an application for 8 events distributed between 2 event groups, with a multiplexing switching interval of 50\(\text{ms}\), sampling time of 100\(\text{ms}\), and a session period of 100\(\text{ms}\). In this scenario, each second of execution invokes 10 times the InOut function pair, 10 times the `task_get_samples` function and 20 times the `switch_mux` function. Using the kernel optimization level (02), the overall overhead of KerMon is 260\(\text{ns}\) per second of execution, i.e., 0.026% of the total execution time. However, even when adding all function overheads, the global overhead of KerMon remains insignificant.

### 4.3 Performance Monitoring

The CARM (see Section 2.4) allows characterizing the behavior of applications and identifying the memory and computation bottlenecks of their execution. It also graphically depicts (in a single plot) the potential of running applications to achieve the maximum attainable performance of a system. For this purpose, the KerMon’s user-space tool `kernelCount` is used to execute the benchmark and fetch the resulting data from the Ring Buffer (see Section 3.4.2) into a file. The `kernelCount` tool configures Scheduler++ to use the event configuration presented in Table 4.3. In brief, Scheduler++ is configured to count all floating-point and memory operations in 2 groups. `kernelCount` also configures the multiplexing switching interval between the groups to 50\(\text{ms}\) and a sampling interval of 100\(\text{ms}\), unless stated otherwise.

Figure 4.3 depicts the CARM rooflines as defined in Section 2.4 for Intel 3770K Ivy Bridge.
4. Experimental Results

<table>
<thead>
<tr>
<th>Event</th>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flops_Sse_Packed_Double</td>
<td>1</td>
<td>Number of [SSE] or [AVX] 128 double precision FP packed uops executed</td>
</tr>
<tr>
<td>Flops_Sse_Packed_Single</td>
<td>1</td>
<td>Number of [SSE] or [AVX] 128 single precision FP packed uops executed</td>
</tr>
<tr>
<td>Flops_Sse_Scalar_Double</td>
<td>1</td>
<td>Number of [SSE] or [AVX] 128 double precision FP scalar uops executed</td>
</tr>
<tr>
<td>Flops_Sse_Scalar_Single</td>
<td>1</td>
<td>Number of [SSE] or [AVX] 128 single precision FP scalar uops executed</td>
</tr>
<tr>
<td>Simd_Packed_Double</td>
<td>2</td>
<td>Number of 256-bit packed double-precision floating-point instructions</td>
</tr>
<tr>
<td>Simd_Packed_Single</td>
<td>2</td>
<td>Number of 256-bit packed single-precision floating-point instructions</td>
</tr>
<tr>
<td>Mem_Retired_All_Loads</td>
<td>2</td>
<td>Number of Load uops retired</td>
</tr>
<tr>
<td>Mem_Retired_All_Stores</td>
<td>2</td>
<td>Number of Store uops retired</td>
</tr>
</tbody>
</table>

Table 4.3: Configuration of the required events for CARM analysis.

The horizontal rooflines represents the peak performance of the following instruction types [38]:

- A scalar instruction (DBL) is a floating-point instruction that only processes one floating-point value. The monitored events Flops_Sse_Scalar_Double and Flops_Sse_Scalar_Single (see Table 4.3) count instructions of this type;

- A single [SSE] instruction processes 2 double-precision or 4 single-precision floating-point values. The Flops_Sse_Packed_Double and Flops_Sse_Packed_Single events count these instructions;

- A single [AVX] instruction processes 4 double-precision or 8 single-precision floating-point values. The Simd_Packed_Double and Simd_Packed_Single events count these instructions.

The diagonal rooflines depicts the peak performance when the execution is dominated by instructions on each memory hierarchy level (L1, L2, L3, DRAM). To avoid cluttering Figure 4.3, it only presents diagonal rooflines for the L1 cache.

Figure 4.3 also shows the performance results for each SPEC CPU2006 benchmark as obtained by KerMon. The marker color attributed to each benchmark represents the dominant instruction type (DBL, [SSE], [AVX]). Figure 4.3 suggests that for the most benchmarks, the predominant floating-point instructions are the scalar (DBL) instructions, with the exceptions of cactusADM, leslie3D and lbm, where [AVX] samples are predominant. In Figure 4.3, it is also possible to verify that calculix has the highest performance among all benchmarks, which marker is located directly under the L1-DBL(MAD) ridge. Figure 4.3 suggests that calculix performs a considerable number of scalar MAD instructions, since calculix predominant type of floating-point instructions are scalar (DBL) instructions and its marker is located between the DBL(MAD) and...
4.3 Performance Monitoring

Figure 4.3: CARM plot of the floating-point SPEC CPU2006 benchmarks. The application color characterization was made according to the dominant instruction type (DBL, SSE or AVX).

DBL(ADD,MUL) horizontal rooflines. However, to perform characterization based on a single point is a simplification of a benchmark real behavior, since it may have completely different behaviors in different intervals that is hidden by the single-point average value. To effectively characterize these benchmarks, further analysis is required.

In order to explore the potential of the features introduced by KerMon, a more thorough analysis for several SPEC CPU2006 floating-point benchmarks is performed. A more fine-grained analysis of the application behavior is important because, in many cases, the same application may show different behavior during the execution. Some of the events that may affect the execution over time include: fluctuations of the pressure on the memory subsystem, different types of instructions (where they may be executed simultaneously, thus creating bottlenecks in different points of the micro-architecture), system changes, execution uncertainties and other non-deterministic behaviors.

Figure 4.4 presents two benchmarking outputs on CARM, namely calculix and tonto. Each dot represents a monitoring sample of about 100ms obtained through KerMon. The dots and lines are colored by the predominant floating-point type (DBL, SSE, AVX). It is possible to notice in Figure 4.4(a) that there is a predominance of scalar (DBL) and AVX instructions on the
4. Experimental Results

Figure 4.4: Evaluation of SPEC CPU2006 benchmarks, namely calculix and tonto, using CARM plots.

Calculix benchmark. It is also noticeable that dots of each instruction type follow closely with the corresponding roofline. This means that calculix limits are the ones defined by the CARM. Since there is a significant amount of dots on the left side of the ridge, calculix is bounded by memory in this section. Although, as can be seen in Figure 4.4(a), it also has a compute-bound section on the right of the ridge point. This means that calculix has also a compute-bound region. Therefore, calculix is comprised of two very distinct behaviors, compute and memory bound.

Figure 4.5: Detail of Figure 4.4(b): CARM plot for the tonto benchmark.

Figure 4.5 is the same plot as Figure 4.4(b), although scaled in order to show more detail and
rooflines. They depict the CARM plot for tonto benchmark. It is possible to see that the dots follow the diagonal rooflines and are mostly on the left side of the corresponding rooflines ridges. It can be observed that the possible bottleneck of tonto benchmark is in the memory subsystem.

Figure 4.6: Evaluation of SPEC CPU2006 benchmarks, namely calculix and tonto, using the CARM over time.

Figure 4.6 shows the detailed performance over time for the calculix and tonto benchmarks. It is possible to see that calculix produces spreader samples, while in tonto, the dots are clustered together. Moreover, in tonto, the predominant instructions have a clear and well-defined alternating pattern between scalar (DBL) and SSE/AVX over time. This hints that the tonto program has two completely distinct functions that execute alternately. Without the CARM plots, the performance/time plots alone do not allow to conclude about the characteristics of an application, namely, if it is memory-bound or compute-bound, although they may provide hints for the application behavior.

Figure 4.7: Evaluation of scalar-dominant SPEC CPU2006 benchmarks, namely gamess_3 and namd, using the CARM.

Figure 4.7 shows the CARM applied to the SPEC CPU2006 benchmarks gamess_3 and namd. For both benchmarks, the predominant type of floating-point instructions are scalar (DBL) instructions. In both cases, the rooflines for L2, L3 and DRAM for scalar (DBL) instructions are also depicted.
4. Experimental Results

Figure 4.7(a) plots the games3 in CARM while Figure 4.7(b) plots the namd execution in CARM. As it can be observed, especially on Figure 4.7(a), the dots follow closely the rooflines, namely the DBL DRAM roofline. Thus, these evaluations show the importance of a more detailed analysis, including one with the rooflines of all levels of the memory hierarchy.

Figure 4.8 shows CARM plots for different multiplexing switching intervals, namely 20ms and 100ms. Since this is a CARM plot composed by 2 event groups as defined on Table 4.3, the sampling interval is the double of the switching intervals, 40ms and 200ms, respectively. As expected, since smaller sampling interval means more samples, Figures 4.8(a) and 4.8(c) present more samples than Figures 4.8(b) and 4.8(d) respectively. By analyzing the CARM plots of the different switching intervals, it is possible to notice that they have a quite similar outline. Although the 20ms plots have spreader points, since a sample with smaller sampling time is more affected by sporadic values than samples with a bigger sampling time, that averages the final result, thus reducing the impact of sporadic values.

Figure 4.9 compares zeusmp and tonto running alone and simultaneously on any LC. In the zeusmp benchmark plots (Figures 4.9(a) and 4.9(b)), it is possible to notice that the samples are more spread, when zeusmp executes simultaneously with tonto. Figure 4.9(c) shows when tonto runs alone, it is mostly L1-bound with a few DRAM-bound samples. However, Figure 4.9(d)...
4.3 Performance Monitoring

Figure 4.9: Comparison of SPEC CPU2006 benchmarks running alone or with other applications, on any LC.

that tonto becomes L3-bound and DRAM-bound, when running along zeusmp. Since tonto is mostly L1-bound when running alone, this means that cached data in L1 and L2 caches is being evicted when it is executed along zeusmp. Since L1 and L2 caches are private core caches, this behavior suggests that both applications are running in the same physical core, thus competing for the private cache levels.

In order to confirm that both applications are executing on the same physical core, the Figure 4.10 is drawn using the data provided by KerMon. It shows the LCs where zeusmp and tonto execute along time. I also shows that zeusmp was always executed on LC 6, while tonto started on LC 5 and migrated to LC 2. In order to check which core belong a LC the OS file /proc/cpuinfo is inspected, that contains information about LCs distribution among cores. It indicated that LC 2 and LC 6 have the same core id, i.e., share the same physical core, therefore, suggesting that both applications are running on the same physical core. This is an example of a scheduling decision that could be improved if the task scheduler considered the information that KerMon provides.

Figure 4.11 compares the performance over time of tonto running alone and with zeusmp. In Figure 4.11(b) it is possible to notice the middle section of the plot, where the performance is
4. Experimental Results

**Figure 4.10:** Distribution of zeusmp and tonto among LCs, when they are executing simultaneously.

**Figure 4.11:** Comparison of tonto benchmark running alone or with other applications, on any LC using the CARM over time.

lower than the edge sections. This behavior corresponds to the tonto section where it executes on the same core as zeusmp:

1. Initially, tonto is executing on a different core than zeusmp. In this section, the performance is similar to when tonto is running alone;

2. After 50s, tonto executes on the same core as zeusmp. It is clearly noticeable that the performance is reduced. Thus, tonto requires more time to execute, as it is possible to see by the length of tonto pattern;

3. After 250s, zeusmp finishes execution, thus tonto runs alone and its performance returns to the nominal performance.
4.4 Energy Monitoring

Figure 4.12: Average package and PP0 power of the floating-point SPEC CPU2006 benchmarks.

Figure 4.12 is generated using data provided by the energy monitoring capabilities of KerMon. Since KerMon provides the energy consumption of each sample and the time interval, the power usage can be obtained with the following equation: \[ Power = \frac{\Delta E_{nergy}}{\Delta T_{ime}}. \]

Figure 4.12 depicts the average package power consumption for each benchmark execution, where the average power over all benchmarks is around 19W. It also depicts the power consumption of the CPU cores, i.e., the PP0 domain (see Section 2.1). As expected, the CPU cores contribute the most for package power consumption, where the average PP0 power over all benchmarks is around 12.6W.

As before, the benchmarks calculix and tonto are further analyzed. Figure 4.13 presents the package power consumption of calculix and tonto over time. In both cases, it can be observed that the AVX instructions usually require more power than the regular scalar (DBL) and SSE instructions. As a result, the tonto benchmark depicts the same alternating pattern that is present in the performance analysis over time (see Figure 4.6(b)).

Figure 4.14 compares the package power consumption over time of tonto running alone and with zeusmp. In Figure 4.14(b) it is possible to notice the three different behaviors:

1. Initially, tonto executes on a different core than zeusmp. In this section, power consumption is significantly higher than the other sections, since the CPU has 2 actives cores;
2. After 50s, tonto executes on the same core as zeusmp. In this section, the power consump-
4. Experimental Results

Figure 4.13: Package power consumption over time. The application color characterization was made according to the dominant instruction type (DBL, SSE or AVX).

Figure 4.14: Comparison of tonto benchmark running alone or with other applications, on any LC, using package power consumption over time.

1. Calculix

2. Tonto

(a) Calculix

(b) Tonto

(a) tonto running alone

(b) tonto running along zeusmp

3. After 250s, zeusmp finishes execution, thus tonto runs alone and power consumption returns to the nominal value.

4.5 Summary

In this chapter, KerMon experimentally demonstrated the insignificant overhead that introduces into the OS task scheduler. Firstly, by comparing the total execution time of the SPEC CPU2006 benchmarks executing on Scheduler++ and on CFS++ scheduling class. Then, the low overhead of KerMon was further asserted by an experimental analysis of the execution time of the most invoked KerMon functions.

Furthermore, the KerMon functionality was also demonstrated for performance monitoring. This demonstration was performed through the analysis and characterization of SPEC CPU2006 benchmarks using CARM. This model with the KerMon performance monitoring data, allowed
4.5 Summary

to reach conclusions about the bottlenecks and characteristics of several applications. KerMon also demonstrated the energy monitoring by correlating the package power/time plots with the performance/time plots. KerMon proved that the provided information allows to provide insightful information about the behavior of the target application, in the performance and energy/power metrics.
4. Experimental Results
5

Conclusions
5. Conclusions

This thesis proposes a solution for in-kernel performance and energy monitoring: The KerMon framework. The nuance of this solution when compared with the existing State-of-the-art is that KerMon allows kernel subsystems and facilities to take advantage of KerMon in order to monitor the performance and energy of applications. Moreover, KerMon is designed and implemented in a way that even the task scheduler (that executes in a very restricted and sensitive context) can invoke KerMon in order to take advantage of the modern CPU facilities, such as HC and RAPL.

KerMon also allows user-space applications to monitor applications. Although, for this purposes there is already existing solutions, namely the Linux perf_events facility and the user-space library. However, since KerMon is integrated with the task scheduler, it must have an insignificant overhead. This condition may appeal for other user-space applications to use KerMon for performance and energy monitoring.

The design of KerMon consists on several components, positioned in the overall computer systems, which main components reside in kernel-space:

- The Raw Events is the component responsible for obtaining performance and energy readings;
- The Event Mux implements time multiplexing in order to measure more events than what is physically possible;
- The Event Logger aggregates these readings into samples and stores them in a buffer;
- The Scheduler++ scheduling class controls the overall KerMon mechanism.

KerMon, beyond being a framework, it already implements an actual benchmarking application. The tools that it provides are the following:

- The libSched++ library that allows user-space developers to fully benefit of KerMon facilities;
- And the kernelCount application that allows non-developers to benchmark applications.

Therefore, with the KerMon component structure, it can be used by kernel developers, application developers and regular users that desire to benchmark applications.

Lastly, KerMon has submitted to experimental tests, where the data obtained through KerMon has analyzed using the CARM in order to identify the bottlenecks that hinders the overall performance. KerMon proved that the provided information allows to provide insightful information about the behavior of the target application, in the performance and energy/power metrics. Experimental tests also allow to assert that KerMon does not affect significantly the overall system performance.
5.1 Future work

In the development of the work presented on this thesis, two branches of possible future work were identified. The first branch is related to possible developments in the KerMon platform itself, namely:

- Add support for performance monitoring on: 
  i) Other CPU platforms, namely AMD and ARM; and
  ii) Other devices, such as GPUs and power supplies.

- Submit the KerMon framework to its integration on the standard Linux Kernel, which would allow anyone to take advantage of KerMon without patching and compiling the Linux kernel.

Since KerMon uses its own scheduling class (Scheduler++), the Scheduler++ can be used as a template for developing scheduling algorithms. Moreover, the also included CFS++ scheduling class, can be used as a control in order to compare the experimental algorithm with currently default CFS scheduling algorithm. Thus, the other development branch is a proposal for a scheduling algorithm that detects resource contention. A resource contention occurs when tasks competitively access the same resources, hindering the performance of both tasks. Resource contention can be detected by using the KerMon performance event monitoring functionality. Therefore, a scheduling algorithm that invokes KerMon, is capable of detecting this contention. Then, it can influence its scheduling decisions in order to minimize the performance loss associated with the resource contention. Developing this scheduling algorithm is more complex than it seems, especially since inner CPU mechanisms are a trade-secret, thus are not available to the public. A system model is necessary to accurately detect resource contention situations. The modeling requires that an extensive benchmarking is performed on several combinations of systems and applications.
5. Conclusions
Bibliography


Bibliography


