Thread scheduling in multi-core operating systems : how to understand, improve and fix your scheduler

Redha Gouicem



Ph.D thesis in Computer Science

Thread Scheduling in Multi-core Operating Systems

How to Understand, Improve and Fix your Scheduler

Redha GOUICEM

Sorbonne Université
Laboratoire d'Informatique de Paris 6
Inria
Whisper Team

PH.D. DEFENSE: 23 October 2020, Paris, France

JURY MEMBERS:

Mr. Pascal Felber, Full Professor, Université de Neuchâtel	*Reviewer*
Mr. Vivien Quéma, Full Professor, Grenoble INP (ENSIMAG)	*Reviewer*
Mr. Rachid Guerraoui, Full Professor, École Polytechnique Fédérale de Lausanne	*Examiner*
Ms. Karine Heydemann, Associate Professor, Sorbonne Université	*Examiner*
Mr. Etienne Rivière, Full Professor, University of Louvain	*Examiner*
Mr. Gilles Muller, Senior Research Scientist, Inria	*Advisor*
Mr. Julien Sopena, Associate Professor, Sorbonne Université	*Advisor*

ABSTRACT

In this thesis, we address the problem of schedulers for multi-core architectures from several perspectives: design (simplicity and correctness), performance improvement and the development of application-specific schedulers. The contributions presented are summarized as follows:

- Ipanema, a domain-specific language dedicated to thread schedulers for multi-core architectures. We also implement a new abstraction in the Linux kernel that enables the dynamic addition of schedulers written in Ipanema.
- a series of performance and bug tracking tools. Thanks to these tools, we show that the Linux scheduler, CFS, suffers from a problem related to frequency management on modern processors. We propose a solution to this problem in the form of a patch submitted to the community. This patch allows to significantly improve the performance of numerous applications.
- a scheduler model in the form of a "feature tree". We implement these features independently in order to offer a new fully modular scheduler. This modular scheduler allows us to study exhaustively the different combinations of features, thus paving the way for the development of application-specific schedulers.

RÉSUMÉ

Dans cette thèse, nous traitons du problème des ordonnanceurs pour architectures multi-coeur en l'abordant sous plusieurs angles: celui de la conception (simplicité et correction), celui de l'amélioration des performances et enfin celui du développement d'ordonnanceurs sur mesure pour une application. En résumé, les contributions présentées sont les suivantes:

- Ipanema, un langage dédié au développement d'ordonnanceurs de processus pour multi-coeur. Nous implémentons également au coeur du noyau Linux une nouvelle abstraction permettant d'ajouter dynamiquement un nouvel ordonnanceur écrit en Ipanema.
- une série d'outils de recherche de bogues de performance. Grâce à ces outils, nous montrons que l'ordonnanceur de Linux, CFS, souffre d'un problème lié à la gestion de la fréquence sur les processeurs modernes. Nous proposons une solution à ce problème sous la forme d'un patch soumis à la communauté. Ce patch permet d'améliorer significativement les performances de nombreuses applications.
- une modélisation des ordonnanceurs sous forme d'un "feature tree". Nous implémentons ces fonctionnalités de façon indépendantes afin de proposer un nouvel ordonnanceur entièrement modulaire. Cet ordonnanceur modulaire nous permet d'étudier exhaustivement les différentes combinaisons de fonctionnalités ouvrant ainsi la voie au développement d'ordonnanceurs spécifiques à une application donnée.

ACKNOWLEDGMENTS

Although this thesis bears my name, it is the result of four years of collaborations, discussions and support from fellow researchers, colleagues, friends and family. Research is not the work of solitary individuals but a collaborative enterprise carried out at one's desk, but also during coffee breaks or family dinners. Without further ado, let's thank everyone that was involved in the making of this work.

First, I would like to thank the reviewers, Pascal Felber and Vivien Quéma, for the time they dedicated to carefully reading and evaluating this thesis. I would also like to thank the examiners, Rachid Guerraoui, Karine Heydemann and Etienne Rivière, for accepting to be part of this jury.

This thesis would not have been possible without the guidance and supervision of Gilles Muller and Julien Sopena. Gilles, your expertise in writing papers and bringing the right people together has made this thesis easier than it could have been. I will strive to maintain the rigor in clearly expressing one's ideas that I learned from working with you. Despite the difficult circumstances of the last two years, you were always there and available when I needed help. Julien, you are one of the people that sparked my interest for systems and for research. Working with you every day has been a pleasure, with its share of ideas, each one crazier than the other, but always relevant. I will never be grateful enough for all the things I learned from you, as a scientist and as a human being. A huge thanks to both of you for these four exciting years. You both taught me how to make good systems research and I hope I will live up to your expectations.

I would also like to address a special thanks to Julia Lawall who was not officially my advisor, but still tremendously helped me improve my writing and critical thinking during these four years.

Another special thanks to my co-authors who accompanied me in all this work, Baptiste Lepers, Jean-Pierre Lozi and Nicolas Palix. It has been great to work with each one of you and share all these productive discussions that improved the quality of my work.

The life of a Ph.D. student can be stressful and make you want to give up. When these times arrive, having a cohesive group of people around you is of the utmost importance. This spirit of cohesion is particularly powerful at the LIP6, and more particularly among the three teams I spent most of my time with, Whisper, Delys and MoVe.

When I started my Ph.D., I was welcomed by extraordinary people that made my integration easy. I would really like to thank Antoine, Gauthier and Damien who have been mentors for me and many other students I think. I hope I was as good a teacher to new students as you were to me.

I would also like to thank all the other Ph.D. students and interns that I had the pleasure to share my days with: Alexandre, Arnaud, Bastien, Cédric, Célia, Daniel, Darius, Denis, Dimitrios, Florent, Francis, Gabriel, Guillaume, Hakan, Ilyas, Jonathan, Laurent, Lucas, Ludovic, Lyes, Marjorie, Maxime, Pierre, Saalik, Vincent, Yoann.[1]

[1] In order not to offend anyone, the list is sorted alphabetically :)

I also particularly appreciated the good atmosphere between senior researchers and students. I would like to thank all of them for the knowledge they passed on to me as teachers and/or as colleagues. To Jonathan, Luciana, Marc, Pierre, Pierre-Évariste and Swan, thanks for everything.

A special thanks to Karine Heydemann who helped me choose which master to apply to when I was a bit lost. I still remember that discussion from nearly 7 years ago now, and I am thankful for it.

In addition to all my friends from the lab, I would also like to thank the other ones that had no idea whatsoever of what *research in computer science* meant but still supported me. Liazid, Manil, Sami, in addition to the laughs, I really know that if I need help in any matter, you would be among the first ones to answer, and I am grateful for that. Thanks to all my teammates from the LSC Handball, with whom we kept a strong bond a team spirit, be it in victory or defeat. Without you, I might have become a little bit crazy at times.

Another special thanks to Ilyas and Maxime for all these moments of laughs and computer science. I am happy to count you among my friends since we met on the benches of University.

And last, but not least, I would like to thank my family for their unwavering support since always. Mom, Dad, I will never be able to repay you for all you did for me. You always pushed me to work hard and I am glad I listened to you. To my brothers Djelloul, Hichem, Mourad, and my sister Amel, thank you for helping me become the man I am now and supporting me all these years.

These acknowledgments have proved to be longer and more serious than I initially intended before starting writing. For once, I have tried not to make jokes for more than five minutes straight, and that is a challenge I would probably not have been able to overcome four years ago. Again, thanks to each and every one of you, and if I forgot anyone, please forgive me, it was probably unintentional.

TABLE OF CONTENTS

1

INTRODUCTION

In 1951, the **Ferranti Mark 1** and the **UNIVAC** computers were released. These two machines were the first commercialized Turing-complete machines, able to execute arbitrary programs in the form of punched tapes. Due to their high cost, such machines were only purchased by government agencies, universities and large companies.[2] They executed a large number of different programs, ranging from scientific computing to accounting and national census.

[2] The most popular computer in the 1950s, the **IBM 650**, cost $500,000 ($4.76 million as of 2020).

These machines were operated by human operators that loaded code and data into the machine, waited for the computation to complete and retrieved the result. These operators were fundamental in their proper functioning as they ensured the order in which programs should be executed. This was the first form of **scheduling**.

Over the years, with the increased processing power of computers, the cost of human operations became important compared to the computing time. To minimize this cost, computer designers sought to make the scheduler a part of the computer and replace human operators. With the introduction of operating systems (OSs) in 1955, the first software schedulers appeared as a central component of resource management in computers.

Since then, a large number of scheduling algorithms have been developed. They target different workloads and performance objectives. Some server applications need to minimize the latency of their requests, while batch applications need to maximize their throughput. On personal computers and smartphones, interactivity is of the utmost importance for users. Embedded devices, on the other hand, can have strict requirements in terms of quality of service and respect of deadlines.

Schedulers have also been influenced by the evolution of the underlying hardware components of computers, be it central processing units (CPUs), memory or input/output (I/O) devices. The appearance of multi-core processors extended the job of the scheduler. In addition to managing when a program should be executed, it must also choose on which core it should execute. Non-uniform memory access (NUMA) architectures, heterogeneous processors and memory hierarchies further complicate the problem of correctly allocating computing resources.

The diversification of workloads and user requirements as well as the dazzling evolution of hardware have always been driving forces in the development of schedulers. Combining both the hardware complexity and the software requirements drastically hardens the decision-making process of the scheduler and increases the complexity of its design.

In this Ph.D. thesis, we study thread scheduling and how it affects the performance of applications. We aim at providing new tools for scheduler developers that help them in their work. Our work can be divided into three axes: **scheduler development**, **performance enhancement** and **application-specific schedulers**.

1.1 SCHEDULER DEVELOPMENT

The first axis, **scheduler development**, aims at easing the development of new schedulers in OSs while maintaining a high level of safety. Developing a scheduler is difficult since it involves multiple areas of expertise: scheduling theory, low-level kernel development and hardware architecture. This highly increases the probability of mistakes in the development process. Errors in the scheduling algorithm can lead to important properties being violated unbeknownst to the developer. Implementation mistakes due to the difficulty of developing in an OS kernel are frequent and can cause system crashes. Lastly, a bad knowledge of hardware architecture can lead to inefficient scheduling policies because of factors such as memory latency or heterogeneous computing units.

Our objective is to remove the need for the kernel development expertise by providing an easy-to-learn high-level domain-specific language (DSL) that will be compiled into C code usable in Linux. Our compiler contributes to the safety of the code by forbidding illegal operations and by automatically generating code such as lock management. The abstractions of the DSL will also encompass the hardware topology to help developers take that into account. We also want to avoid algorithmical errors by allowing formal verification of scheduling properties to be semi-automatically performed through our DSL.

In addition, we also provide a new feature to Linux, Scheduler as a Kernel Module or `SaaKM`. With this feature, we enable users to insert, at run time, new scheduling policies. Each application can then decide which policy to use. The C code generated by our Ipanema compiler is compatible with `SaaKM`.

Thanks to `SaaKM` and our DSL, we develop and test multiple schedulers, such as a simplified version of the Linux scheduler that performs similarly to the original on the set of applications we evaluate. We also

develop a version of this scheduler proven to be work-conserving that outperforms the original on some applications.

1.2 PERFORMANCE ENHANCEMENT

The second axis, **performance enhancement**, aims at helping scheduler developers finding *performance bugs*. These bugs do not cause crashes but silently eat away at performance. They are therefore hard to notice and solve. There are two ways of detecting such bugs: producing a better scheduler that does not suffer from this bug, or using profiling tools that highlight the bug.

We design monitoring tools to efficiently profile the behavior of schedulers, and visualization tools to easily track performance bugs and identify their origin. These tools enable us to record scheduling events at run time without disturbing application behavior and with a high resolution. They also allow us to visualize specific data such as the CPU usage of each core, their frequency or all scheduling events.

With these tools, we detect a new problem, **frequency inversion**, that stems from modern per-core dynamic frequency scaling and the scheduler unawareness of CPU frequency in its thread placement decisions. This problem triggers situations where high frequency cores are idle while low frequency cores are busy. After finding the root cause of this problem, we propose two solutions for Linux to solve the problem and thoroughly evaluate them. The best solution has been submitted to the Linux community for review.

In addition, we also provide a detailed analysis of the behavior of the frequency scaling algorithm on multiple CPUs. This analysis was possible thanks to our monitoring tools and further strengthens our belief that schedulers must account for the frequency of cores.

1.3 APPLICATION-SPECIFIC SCHEDULERS

The third and last axis, **application-specific schedulers**, aims at helping software developers use the best possible scheduler for their application instead of always using the default scheduler of the OS. Even though general-purpose schedulers try to be generic and offer good performance for most workloads, they are not able to always offer the *best* performance. This is mainly due to two major problems: the structure and size of their code, and the necessary configuration.

The structure and size problem is prominent in general-purpose schedulers such as Linux's CFS. They tend to be large, with a considerable number of features. These features also tend to be intertwined, be it in terms of code or impact. This increases the likelihood of safety or performance bugs, and hardens the maintenance of the code.

The choice to be generic also creates a fundamental problem: since the expectations of users differ and are sometimes conflicting, it is

impossible to always satisfy everyone. To overcome this problem, most general-purpose schedulers are configurable through static configurations at compile time or dynamically at run time. Due to the difficulty of finding the correct configuration for a given workload, most users just use the default values provided by their OS. For more advanced users, there also exist a multitude of user-provided configurations and tips on system administrator forums on the internet.

We propose to start building an actual modular scheduler from scratch. To do so, we develop a **feature model** of a scheduler where each feature is implemented independently, making it easy to extend. This model also allows for an evaluation of each feature separately from the others. With such an evaluation, we propose methodologies to find the most adapted features for a given workload. Finally, we propose methodologies to build application-specific schedulers from this data.

1.4 OUTLINE

The remaining of this document is organized in five chapters. Chapter 2 presents the technical background and the state-of-the-art on thread scheduling. It lays off the needed knowledge to understand our contributions. Chapter 3 presents our first contribution, **Ipanema**, a DSL for schedulers. We present the design of the language, its tool chain and an evaluation of multiple policies we implement in Ipanema. Chapter 4 presents our second contribution, the identification of a new problem, **frequency inversion**, and strategies to solve it. This problem stems from modern implementations of frequency scaling on CPUs and current general-purpose schedulers' behavior during thread placement. We present the problem through a detailed case study and propose two strategies to solve this problem in Linux that we extensively evaluate on multiple machines. Chapter 5 presents our last contribution, a **feature model of a thread scheduler**. This model is designed with the objective of evaluating scheduler features individually in order to design application-specific schedulers. We present various methodologies that help finding the best suited scheduler for a given application. Finally, Chapter 6 concludes this thesis with a summary of our work and contributions, and discusses future work and perspectives.

2

THREAD SCHEDULING

This chapter aims at defining basic concepts regarding thread scheduling used throughout this thesis, and giving the reader the necessary technical background to understand our contributions. The thread scheduler is the component of a system that manages the allocation of computing resources to software. In this chapter, we first define the hardware and software resources managed by thread schedulers. We then describe in details what are thread schedulers and how they operate. Finally, we describe how thread scheduling is implemented in various systems, with a focus on Linux.

2.1 EXECUTION ENTITIES

In operating systems theory, multiple concepts describe software and how it should be executed: threads, processes, tasks, ... Computer scientists tend to mix these terms, usually without loss of meaning in their context. However, in this thesis on scheduling, using one term instead of another would be a mistake and lead to misunderstandings. The following definitions will be used throughout this thesis to avoid such misunderstandings. Figure 2.1 summarizes the hierarchy between each decribed entity.

THREAD. A thread is the entity that executes instructions, and the smallest entity manipulated by the thread scheduler. It is usually represented as an instruction pointer (IP), a stack pointer (SP) and registers. When allocated a computing resource by the scheduler, a thread executes the instruction pointed to by its IP. Threads can be handled at the kernel or at the user level. Kernel threads are managed by the OS's kernel while user threads are manipulated by user level programs such as language runtimes or libraries. Ultimately, user level threads are mapped to kernel threads in order to be scheduled by the OS's kernel. In this thesis, the term thread will be used for kernel threads unless otherwise stated.

PROCESS. A process represents a set of resources comprising a memory mapping, file descriptors, sockets, ... A process is used by at least one thread. If multiple threads share a process, communication

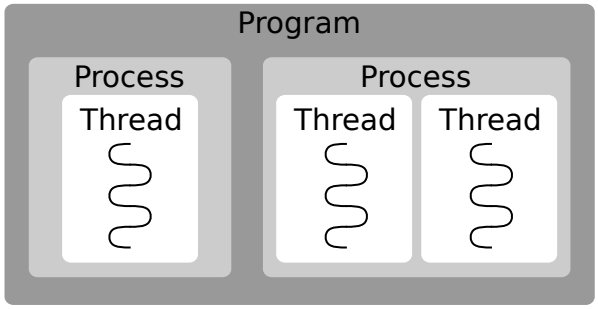


Figure 2.1: Illustration of a program with multi-threaded processes.

is simplified thanks to the shared resources. Communicating between processes, on the other hand, is more heavyweight and relies on the use of inter-process communication mechanisms provided by the OS.

PROGRAM. A program, or application, is a set of one or more processes, containing one or more threads, that cooperate in order to fulfill a common objective. For example, a web browser that spawns one process per tab is still considered as a single program.

> TASKS
>
> The term **task** has various meanings depending on the context. It is used as a synonym to thread or process in Linux. In the Java language, it is either a synonym for thread (when using the `Thread` class) or a unit of work that can be performed by any thread (when using the `Executor` interface). Due to this ambiguity, the term task will not be used in this thesis.

2.2 HARDWARE RESOURCES

The thread scheduler is the software component that manages the computing units in a system. These computing units have immensely evolved since the beginning of computing. Figure 2.2 shows the hardware topology of a modern machine. In this topology, there are execution units (cores), memory accelerators (caches) and main memory. Some of these components are shared while others are exclusive. All these interactions make the allocation of computing resources a complex job for the scheduler. We present the different hardware components, their possible interactions and impact on the thread scheduler.

2.2.1 *The Core: an Execution Unit*

A core, also called a hardware thread, is the smallest computing unit found in a processor chip. It executes the instructions of a single thread

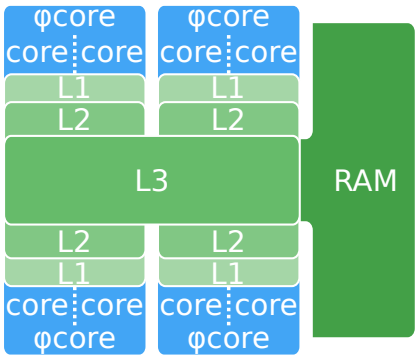


Figure 2.2: A single node 8-core machine, with two threads per physical core (*φcore*) and three levels of caches.

of execution at a time. This is the actual resource the thread scheduler manages, allocating it to software threads for given periods of time.

A core is usually composed of a set of registers and computing units. The registers store data used by the running thread as well as metadata used to manage the thread, such as its instruction pointer or its stack pointer. The computing units perform various computations such as arithmetic operations (arithmetic logic unit (ALU)), floating-point operations (floating-point unit (FPU)) or address calculations (address generation unit (AGU)).

CHIP-LEVEL MULTIPROCESSING. Since the 1950s, a large number of innovations in computer architecture improved the individual performance of cores. Instruction-level parallelism techniques, such as instruction pipelining, speculative execution [19] or out-of-order execution [80], have no impact in terms of scheduling, besides executing instructions more efficiently. Thread-level parallelism techniques, however, change the way the thread scheduler operates by allowing multiple threads to run at the same time. Chip-level multiprocessing (CMP) implements thread-level parallelism by integrating multiple cores into a single chip. Each core is identical and can therefore execute a different thread independently. With CMP, the scheduler can now choose to run multiple threads at the same time.

SIMULTANEOUS MULTITHREADING. Early on, computer architects noticed that all computing units were not used at the same time, leading to a waste of computing resources. To solve this, they introduced simultaneous multithreading (SMT) [160]. The idea of SMT is to run multiple threads on the same **physical core** by duplicating part of the hardware, mainly registers. These duplicates are called **logical cores**. For example, Intel®'s most common implementation of SMT, Hyper-Threading [85], allows to run two threads per core, i. e. for n physical cores, $2n$ logical cores are available. In this thesis, we will use the term **core** to refer to logical cores, since schedulers place threads

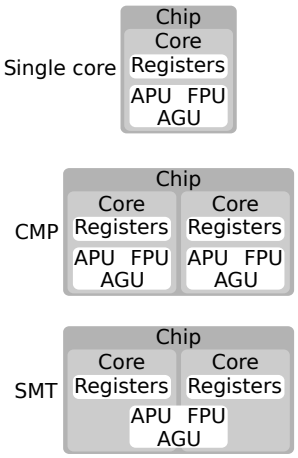


Figure 2.3: Thread-level parallelism implementations.

on this kind of core. SMT has an impact on thread scheduling because some operations will be impossible to perform simultaneously by two cores sharing the same computing units. Figure 2.3 summarizes the difference between CMP and SMT.

ASYMMETRICAL ARCHITECTURES. Some processor chips also feature cores with different capabilities. These **asymmetrical architectures**[3] can have cores specialized in a certain type of computations like video or audio decoding [71, 86, 145]. Another use of this type of architecture is energy management on embedded devices. For example, the ARM® big.LITTLE architecture [11] has a set of low power and low performance cores (LITTLE) and a set of more powerful and power-hungry cores (big). Using cores from one or the other set has considerable impact on performance and energy consumption.

[3] Also called heterogeneous architectures.

The implementation of dynamic voltage and frequency scaling (DVFS) technologies can also be seen as a form of asymmetry. Indeed, if each core can run at a different frequency, they all have a different processing power available. This will be discussed in more details in Chapter 4.

2.2.2 *Caches: a Memory Access Accelerator*

When cores perform their computations, they usually work with data available in the main memory. Accessing this memory is a frequent operation in software, thus leading researchers to improve memory access efficiency. A widespread way of doing so is to place small hardware caches between cores and main memory. Caches are faster than main memory but are more expensive. The goal is to exploit two properties: temporal locality [182] and spatial locality [109].

Temporal locality specifies that recently accessed data has a high probability of being accessed again in the near future. To exploit this property, caches are used to keep recently accessed data close to cores using them.

Spatial locality specifies that when a piece of data is accessed, neighboring data has a high probability of being accessed in the near future. To exploit this property, when a piece of data is loaded into a cache, adjacent data is also loaded for future accesses. As a result, computer architects design memory hierarchies with fast and small caches close to cores, and slower but larger caches farther from cores and closer to memory. This is visible in Figure 2.2 with three levels of caches available.

Caches can also be used to easily share data between cores. In our example, there are two levels of caches exclusive to a single physical core (L1 and L2) and one level that is shared between all cores (L3). Sharing is beneficial when multiple cores access the same data, removing the need to load it from main memory again and

again. However, if all cores use different data, they compete for the same cache locations, leading to data evictions, and subsequent main memory accesses. This phenomenon is called cache thrashing. Cache sharing or thrashing has an impact on scheduling decisions since different thread placements can induce more or less sharing and thrashing.

2.2.3 *Main Memory: a Potential Bottleneck*

Even though caches reduce the number of memory accesses, main memory is still a critical resource. Some accesses cannot be cached and some access patterns make caching an inefficient optimization. For these reasons, main memory is still a potential performance bottleneck.

SYMMETRIC MULTIPROCESSING. The coupled use of a multi-core design (CMP and/or SMT) and cache hierarchies connected to a single main memory is called symmetric multiprocessing (SMP).[4] These architectures use an *interconnect* to link all their components, from caches to main memory and I/O devices.[5] One main characteristic of these systems is that accessing memory has the same cost, regardless of the core doing the access. These SMP architectures greatly improve performance by allowing multiple threads to run concurrently with fast memory accesses.

With the rising number of cores, the number of requests to main memory has drastically increased. At the same time, technologies behind main memory evolved more slowly than those behind processors. With more than a dozen cores, main memory cannot keep up with the number of requests and respond in an acceptable time.

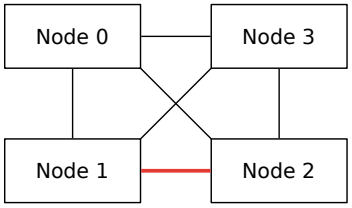
[4] Multiple definitions exist, including or excluding caches, but we will use this one for the remainder of this thesis [43].

[5] The interconnect can be a shared system bus or a point-to-point interconnect.

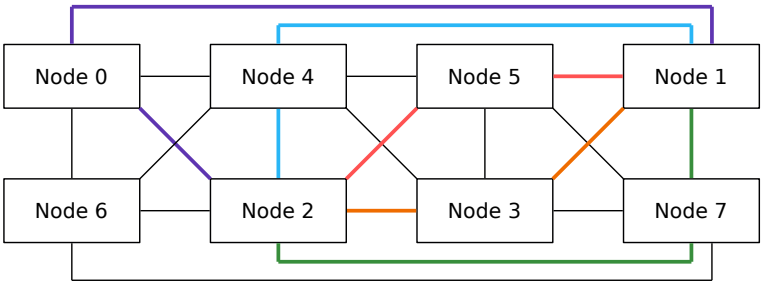
NON-UNIFORM MEMORY ARCHITECTURE. One way to solve this problem is to apply on main memory the same recipe that was applied on cores: duplication. Instead of having a single main memory that serves all cores, the machine is split into **nodes**, as illustrated in Figure 2.4. A node can be seen as an SMP system, with a set of cores, caches and its own main memory. The objective is to limit the number of cores that access the same main memory concurrently. Ideally, each node is a closed system where cores only access the main memory of their node, i. e. **local memory**.

Nonetheless, a system where each node is fully isolated is hardly possible. Some data must be shared between nodes for multiple reasons, including data from the OS or large-scale applications that have threads spanning multiple nodes. Nodes are therefore connected to each other to allow cores on a given node to access the memory of another node, i. e. **remote memory**.

Remote accesses are not as fast as local accesses since they must go through additional interconnections. These non-uniform memory



(a) 4-node Intel® Broadwell machine (1 hop).



(b) 8-node AMD® Bulldozer machine (2 hops).

Figure 2.4: Topologies of two NUMA machines. Shortest routes from node 1 to node 2 are highlighted.

access (NUMA) architectures come in multiple flavors, with different degrees of uniformity between access times. Figure 2.4a shows a fully connected NUMA architecture where all remote accesses roughly take the same time to complete under the same level of contention. Figure 2.4b, on the other hand, shows a topology where nodes are not fully connected. If a thread on node 1 wants to access data on node 2, the shortest routes are two hops long, going through one node on the way. The five possible routes are highlighted in Figure 2.4b. The longer the distance between two nodes, the longer it takes to access memory. The use of such routes can lead to contention on the interconnect, mitigated by routing algorithms implemented in hardware. Software can also help mitigation by placing threads and memory on the best nodes possible.

2.3 THREAD SCHEDULING

We now have a clear picture of the computing resources, i. e. cores, and the entities that use them, i. e. threads. In order to manage these cores and allocate them to threads, we need a component that will handle this job. The thread scheduler is this component. The design of a scheduler can be expressed as the answer to four questions:

- WHICH thread should run?
- WHEN should it run?
- WHERE should it run?
- WHY should it run?

The thread scheduler's job is to answer these four questions. In this section, we first provide some early history of scheduling. We then detail the generic model commonly used to define threads, the ins and outs of each question, and how schedulers answer to these questions.

2.3.1 *Early History*

In the early days of computing, mainframes did not have OSs. Operators manually inserted their programs and data in the form of punch cards or magnetic tapes, waited for the computer to perform all needed computations, and retrieved the results printed by the machine. In the 1950s, as mainframes became more and more fast, the cost of human operations became important compared to the computing time. Pioneers of computing set out to solve this problem by automating these operations: OSs were born.

The first OSs[6] used a **monoprogramming** paradigm, where a single application could run at a time, with no means to switch between applications until their completion. The completion time of a program depends on its own execution time as well as the completion times of the programs running before it. The order in which programs would be scheduled was manually chosen by a human operator.

[6] The first OS, GM-NAA I/O, was released in 1955 by General Motors for the IBM 704 computer [146].

Later on, in the 1960s, **multiprogramming** OSs made their apparition, with IBM's OS/360 leading the way. The MVT variant of this OS, released in 1964 for the large machines of the System/360 family, introduced **time-sharing**. This allowed multiple programs to run at the same time, as well as interactive usage of computers. Developers were now able to write programs with a text editor while batch jobs were running in the background. The software component that governs how computing resources are shared between application, the scheduler, was born.

2.3.2 *The Thread Model*

As stated previously, a thread is the smallest schedulable entity in a system. During its lifecycle, it goes through multiple states that depend on the code it is running. It is in a RUNNING state when using a computing resource, READY when waiting for a computing resource, or BLOCKED when waiting on an external resource such as reading an I/O device. This behavior is best modeled as the finite state machine shown in Figure 2.5. All transitions of this finite state machine can lead to the intervention of the scheduler. *Election* determines **which** thread should run next on a core, while *preemption or yield* determines **when** a thread should stop using a core. These transitions can be triggered by the application, e. g. by yielding the CPU through a system call, or by the scheduler itself.

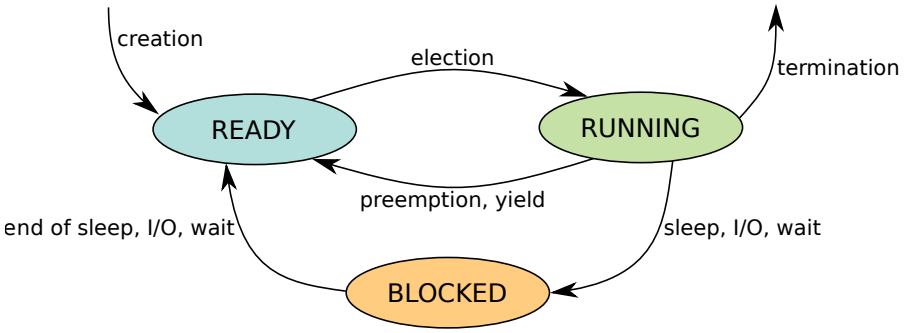


Figure 2.5: Finite state machine modeling a thread.

The decision to change the currently running thread can be taken at various times. **Non-preemptive** schedulers[7] are triggered only when the running thread relinquishes the use of the processor, be it due to a blocking operation, e. g. an I/O operation, or a willing yielding, i. e. calling the `yield()` system call. In this setup, the scheduler only decides which thread should run, but not for how long. This will depend on the selected thread's behavior. This type of scheduler was used in old OSs but was dismissed because malicious or bugged threads could hang the machine by never blocking, e. g. executing an infinite busy loop. However, non-preemptive scheduling presents the benefit of simplicity. Applications do not need to worry about being interrupted, easing the development process. Non-preemptive schedulers are still used in controlled setups such as embedded systems where all running applications are known.

[7] Also called cooperative schedulers.

Most modern general-purpose schedulers allow thread **preemption**. This means that the scheduler is not only called when the running thread stops using the processor, but also when the OS wants to. For example, the scheduler can be triggered when the running thread exits a system call or after handling a hardware interrupt. This design gives more opportunities of scheduling out a malicious or bugged thread that would otherwise hog a core. Even in a sane environment, preemption enables the scheduler to schedule a waking up thread if its priority is higher than the currently running thread's priority.

Another usage of preemptive scheduling is the implementation of **time-sharing** policies. In order to allocate the processor to a thread for only a given period of time, time-sharing schedulers usually periodically trigger a timer interrupt. During each interrupt, or **tick**, the scheduler evaluates if the currently running thread should still be scheduled until the next tick. If not, the election mechanism is triggered to choose a new thread to run. This is an essential feature of interactive systems like personal computers or smartphones. It is also essential in shared servers where different clients should be able to use the processing resources fairly.[8]

[8] Fairness is not a synonym for equality, it can be weighted depending on multiple factors.

2.3.3 *Election*

Originally, CPUs only had a single core, and the scheduler was only responsible for deciding **which** thread should run on the core and **when**. Different strategies can be put in place for this purpose, such as fairly assigning the same time to every thread, or prioritize some threads over others. Usually, general-purpose OSs choose fair approaches because the main goal is to provide good performance for every application. However, the behavior of each application is different, and treating each thread equally might not be in the best interest of overall performance.

Applications are traditionally classified into two categories: *I/O-bound* and *CPU-bound*. *I/O-bound* applications frequently perform I/O operations and relinquish using the processor. I/O operations can be accesses to storage or network, or waiting for a keyboard press from the user. These applications usually require to use the CPU frequently but shortly, and aim at low latency. *CPU-bound* applications, on the other hand, tend to mostly use the processor without performing I/O operations. They usually require long consecutive periods of time to use the CPU to minimize the cost of context switching between threads and improve cache usage, and aim at high throughput.

In the real world, applications can exhibit both behaviors: an application can have different phases that are either CPU- or I/O-bound. For example, a web browser is I/O-bound when waiting for user input (e. g. a URL) and CPU-bound when processing the content of a webpage to display it to the user. The duality of these behaviors can also be seen in multithreaded applications where some threads are I/O-bound while others are CPU-bound.

With these two classes of threads in mind, the thread scheduler of a general-purpose OS must try to satisfy all threads equally. This is done through various algorithms and heuristics that, depending on the thread's behavior, determine its needs and the best decision to fulfill them. For example, I/O-bound threads will have a higher chance of being selected than CPU-bound threads because they use the CPU for very short periods of time. However, the number of threads to satisfy and their differing needs make it a hard job for the scheduler to fairly provide the best possible performance for all threads.

In a specific OS, as opposed to a general-purpose OS, fairness is not necessarily a concern. For example, real-time OSs tend to implement unfair strategies to ensure that operations are performed in a timely manner deterministically. For example, the *Electronic Brakeforce Distribution* technology [26] in modern cars computes the force to apply on each wheel of the vehicle depending on multiple factors (force on the pedal, speed, road condition, ...). If this computation is too slow, the data collected by the sensors becomes obsolete, and braking not as efficient as it should have been. Real-time system engineers therefore

choose a *deadline* before which the computation must complete at all costs. The thread scheduler of computer systems running such applications does not care for fairness between threads, but instead care for the respect of all deadlines.

2.3.4 *Placement Management*

With multi-core architectures, the thread scheduler must also decide **where**, i. e. on which core, a thread should be executed. This new role adds a whole new complexity to thread scheduling, not only because of these new resources to allocate, but also because of the way in which these resources are interconnected. While cores are exposed as independent resources, they still share hardware at different levels, as described in Section 2.2. These resources include computing resources (SMT), caches, main memory and interconnects. When shared by multiple cores, they are subject to contention. Scheduling decisions directly affect the level of contention on every shared component, placing the thread scheduler at the center of resource management.

Resolving the contention problems of these modern architectures at the scheduler level is done in various ways. One way to solve cache contention is to spread threads across cores sharing the minimum number of caches. This will improve the performance of the applications if threads a thrashing each other's data from the caches. However, when threads share data, doing this could reduce the performance and increase the contention on interconnects and memory controllers. This is due to the fact that data would need to be accessed from memory and the traffic due to cache coherence protocols will increase. Similarly, taking advantage of shared caches in a collaborative application might induce contention on the shared SMT hardware, caches, as well as on the local memory controller.

As is the case with time management, placement management must take advantage of multiple hardware technologies aimed at improving performance, with each improvement potentially deteriorating the performance of another piece of hardware. General-purpose OSs must find the best compromise in order to achieve the best overall performance for all running applications.

Heterogeneous architectures also introduce complexity in terms of thread placement. As presented earlier, asymmetrical architectures have cores with different capabilities, and the scheduler must take them into account. For example, on ARM® big.LITTLE architectures, the scheduler must decide to favor either performance or energy saving by placing threads on big or LITTLE cores.

For all these reasons, thread placement is an essential component of the scheduler. It is also a very complex one due to its particularly close relation to hardware.

2.3.5 *Scheduling Properties*

As described on various occasions previously, a scheduler ensures a set of properties that affects its performance. A scheduler developer must ask himself **why** he is developing a scheduler to deduct which properties must be enforced. A property is not inherently good or bad, it is the targeted applications and use cases that make a property good or bad for a given setup.

LIVENESS. **Liveness**[9] ensures that a thread requiring computing resources, i. e. a READY thread, will get access to a computing resource in finite time [148]. This is desirable in general-purpose OSs where no thread should be starved from using the processor. Note that this does not mean that the application necessarily makes progress, as poorly coded applications might get stuck in busy loops or livelocks. From the point of view of the scheduler, this is still considered as progress.

[9] Also called freedom from starvation.

FAIRNESS. **Fairness** ensures that all threads are given the same amount of computing resources. Again, it is usually a desirable property in a general-purpose OS scheduler. The level of fairness between threads can also have a large impact on performance. Being "too" fair would mean context switching between threads more frequently, thus wasting valuable computing resources. Fairness should not be confused with equality: all threads must not have the same allocated CPU time, it also depends on the thread's requirements.

Fairness can be provided by **proportional share** schedulers such as Fair Share Schedulers [76, 92] or Lottery Scheduling [178]. These schedulers are largely inspired or influenced by network queueing algorithms [15, 46]. Patel *et al.* [135] improve fairness with scheduler-cooperative locks. Zhong *et al.* [194] do a similar thing in virtualized environments.

PRIORITY. Some systems schedule threads by strict **priority**: the thread with the highest priority must run before all other threads. With this type of scheduler, high priority threads have a high probability of running quickly and perform their work uninterrupted by other threads. In these priority-based schedulers, when multiple threads share the same priority, there must be a way to select which one should run. This choice is usually done in a round robin fashion.

Unix-based systems expose priority through the *nice* value, although it is sometimes only used as a parameter in proportional share schedulers. Priority-based scheduling is also proposed for hard or soft real-time systems [74, 104, 154].

RESOURCE SHARING. On systems with SMP or NUMA architectures, contention on shared hardware can quickly become a perfor-

mance bottleneck. To address this issue, schedulers can take the usage of such shared resources into account and schedule threads in a way that minimizes contention. This can be done by avoiding to run two threads that use the same hardware resource at the same time, or by placing threads on distant cores to avoid resource sharing altogether. On the other hand, sharing can be beneficial in some cases. For example, two threads working on the same data benefit from sharing caches, thus diminishing the number of memory accesses performed.

Multiple approaches were proposed regarding different shared resources. Systems featuring SMT have been studied to minimize contention due to this technology [7, 134, 161]. Various work focus on memory contention [9, 155, 188] or shared caches [57, 58]. Other approaches propose different heuristics to detect and avoid contention or use beneficial sharing [120, 162, 163, 192, 195].

INTERACTIVITY. A large number of applications require good performance in terms of **latency** or **interactivity**. For example, user interfaces must react quickly to inputs so that the end user does not notice lag. Databases and servers also have request latency requirements to fulfill in order to be responsive.

Automated approaches that estimate the requirements of applications on the fly based on system observations have been proposed [14, 42, 50, 147, 164]. Redline [190] improves the performance of interactive applications by allowing users to specify requirements to the scheduler without modifying the application. Other solutions propose to provide an application programming interface (API) to applications so that they can communicate their requirements to the scheduler at run time [3, 101, 150].

WORK CONSERVATION. On multi-core systems, another interesting property is **work conservation**. A work-conserving scheduler leaves no core idle if there is a READY thread available on the system. This can be seen as either a good or bad property, even from a purely performance-oriented point of view. Using all cores might increase contention on shared resources [56], but it also increases the computing power available for threads. The benefits of this property is application-specific. Most general-purpose schedulers try to achieve work conservation, although not necessarily at all times due to the induced scheduling cost.

REAL-TIME. In real-time contexts, a usually desirable property is the **respect of deadlines**. Real-time applications perform tasks that must complete before a given time called a *deadline*. Real-time scheduling algorithms, such as Earliest Deadline First (EDF) [110] or deadline monotonic scheduling [12] enforce deadlines for applications if the sys-

tem is sized correctly, i. e. it is possible to run all applications without exceeding deadlines.

ENERGY. Embedded systems could also require to keep a **low energy consumption**. This can be solved at the scheduler level at the expense of performance. This type of systems can also disable clock ticks to save energy on idle cores. The scheduler must therefore be adapted to work on these **tickless** systems.

Prekas *et al.* [141] improve energy proportionality, i. e. the quantity of energy consumed compared to the work performed. Merkel *et al.* [120] use co-scheduling and dynamic voltage and frequency scaling (DVFS) to improve performance and save energy.

PAIRING PROPERTIES. Some of these properties are contradictory with one another. For example, it is not possible to be work-conserving and to lower energy consumption by not using some cores. Similarly, fairness can be unwanted in a real-time context, where respecting deadlines is of the utmost importance.

Conversely, pairing some properties can be highly beneficial for some workloads. In a real-time context, reducing contention over shared resources can allow threads to respect their deadlines more easily [18].

2.3.6 *The Place of Thread Scheduling in a Computer System*

The operating system is the interface between hardware resources and software. It is responsible for the management and allocation of hardware resources to software that require them. Those resources include, but are not limited to, memory devices, input/output devices and cores. OSs are commonly divided into two parts, kernel and user spaces, each having its own address space.[10] The boundary between both spaces depends on the chosen kernel architecture: a monolithic design will embed all OS services in kernel space, while a microkernel approach will push the maximum number of services away from kernel space towards user space.

[10] Single address space approaches exist, such as the Singularity OS [79]

Thread scheduling can be performed at different levels of the OS, and even outside of it. Most general-purpose OSs, such as Linux, FreeBSD or Windows, implement scheduling at the kernel level. Another approach would be to implement it as a user space service of the OS. Thread scheduling can also be done by user applications themselves. This is the case with multiple threading libraries like OpenMP [44]. Some languages, most notably Go [48], expose lightweight threads (`goroutines` in Go) that are scheduled by the language's runtime. These user level schedulers either exploit capabilities offered by the underlying OS scheduler to manage thread

> *"A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality."*
> — ***Liedtke Jochen [107]***

ordering and placement, or add a new layer of scheduling above the OS scheduler.

Finally, in a similar way, virtualized environments can also add a layer of thread scheduling. The hypervisor allocates a number of virtual CPUs (vCPUs) to each hosted virtual machine (VM), and each VM sees these vCPUs as physical CPUs. Therefore, the hypervisor schedules vCPUs on physical CPUs, while each VM schedules threads on vCPUs.

From these layered schedulers arise multiple problems. First, one layer does not necessarily know that other layers exist, most notably with virtualization. This means that schedulers in VMs may take decision with incorrect informations relayed by the hypervisor unknowingly. Second, each layer may take decisions independently, leading to conflicts between each layer. These challenges make it difficult to achieve the best possible performance, with multiple schedulers acting without communication between them.

> CLUSTER SCHEDULING
>
> Computation can also be managed at the cluster level. The goal of this cluster scheduler is to place jobs on machines. Each machine will then perform scheduling locally. This topic will not be covered since it is out of the scope of this thesis.

2.4 SCHEDULING IN THE LINUX KERNEL

With this understanding of the general principles of thread scheduling, we can now dive into the description of production schedulers used every day. In this thesis, all contributions are implemented in the open source GNU/Linux environment. More precisely, the scheduler subsystem is located in the Linux kernel. We provide a tour of this subsystem in order to better understand the contributions of this thesis.

2.4.1 *Overview*

The Linux kernel, and the distributions that use it, are Unix-like systems that implement a large part of the Portable Operating System Interface (POSIX) [81] standards. More precisely, in terms of thread scheduling, the POSIX.1 standard defines scheduling policies, thread attributes and system calls that allow users to interact with the scheduler.

SCHEDULING POLICIES. The standard requires three scheduling policies, i. e. scheduling algorithms, to be implemented: `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`. Through specific system calls, threads

can change from one policy to another during their execution. As their name suggests, `SCHED_FIFO` and `SCHED_RR` respectively implement First-In First-Out and round-robin algorithms to select which thread to run. They are mainly used for real-time applications. The `SCHED_OTHER` policy, is implementation-defined and can be whatever the OS developers want it to be. It is usually used to implement a fair policy in general-purpose OSs. More details on these policies will be provided in the following sections.

THREAD ATTRIBUTES. Two mandatory thread attributes are defined by the POSIX standard: the thread **priority** and the **nice** value. These two terms have very similar meanings and are often confused. However, they define two different concepts that are important to dissociate.

Priority reflects the importance of a thread, and therefore the relative urgency of a thread to be scheduled compared to others. The higher the priority, the most chance a thread has to run. Priority can evolve over time, as seen fit by the scheduler, to reflect a thread's behavior. For example, as previously stated, a widespread decision is to raise the priority of I/O-bound threads while reducing the priority of CPU-bound ones.

The **nice** value is a handicap a thread can impose on itself regarding the use of the processing resources. A high nice value means that the thread is not in great need of using the processor and prefers to let other threads use it instead. This value is thus a hint given by threads to the scheduler that specifies their scheduling needs. Threads can increase their *nice* value to communicate their lack of urgency to use the processor to the kernel.[11]

[11] On most systems, any thread can increase its nice value, but decreasing it requires elevated privileges.

SYSTEM CALLS. The interaction between threads and the scheduler is also driven by multiple system calls. The only mandatory one directly related to scheduling is `sched_yield()`. It allows a RUNNING thread to relinquish its ownership of the processor resource. Other system calls transition a thread to a BLOCKED state, most notably those accessing I/O devices such as storage or network devices. Such system calls are defined by the POSIX standards, but not directly as scheduler-related system calls. Their impact on scheduling is more a side effect needed for performance than a direct interaction. If they did not trigger scheduling events, BLOCKED threads would keep the ownership of the processor even though they cannot use it. There are also other system calls that remain optional, such as the ones used to move from one scheduling policy to another or to manage real-time policies.

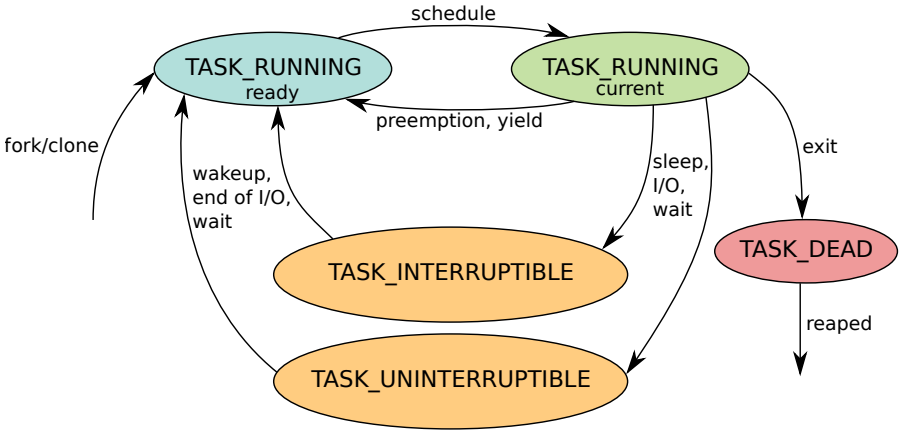


Figure 2.6: Thread states in Linux.

2.4.2 *Threads*

In the Linux kernel, each thread is represented by a set of variables embedded in a structure (`struct task_struct`). These variables include the thread's address space, a pointer to its parent, file descriptors, pending signals, as well as statistics such as its execution time or the number of bytes accessed through I/O devices. From now on, we only focus on scheduler-related attributes.

From the scheduler subsystem's point of view, a thread is defined by its *state*. As seen in Section 2.3.2, these states can be seen as states of a finite state machine while scheduler functions can be seen as transitions. Figure 2.6 shows the finite state machine used in Linux. Although similar to the three-state thread model presented earlier, there exists some differences between the generic and the Linux model.

Runnable threads in the runqueue are in the `TASK_RUNNING` state, tagged as `ready` in the figure. This is the equivalent of the READY state. The currently running thread is also in the `TASK_RUNNING` state, tagged as `current`. This is equivalent to the RUNNING state. Transitioning from the former to the latter is done when the `schedule()` function is called, while the reverse transition is due to preemption (exhausted time slice or higher priority thread available) or yielding.

Threads in the `TASK_INTERRUPTIBLE` state sleep until they are woken up, e. g. end of I/O or reception of a non-masked signal, whereas threads in the `TASK_UNINTERRUPTIBLE` state behave in the same way except they ignore signals altogether. Going into one of these states means that a RUNNING thread went to sleep or is waiting for a resource (I/O, lock) to be available. When the resource becomes available, the thread goes back to a READY state, ready to use the CPU.

The `TASK_DEAD` state is a temporary state used at thread termination. When in this state, the thread will no longer be executed. Its metadata are kept in memory until its parent thread *reaps* it. This is necessary to pass the return value of a terminated thread or to know how this

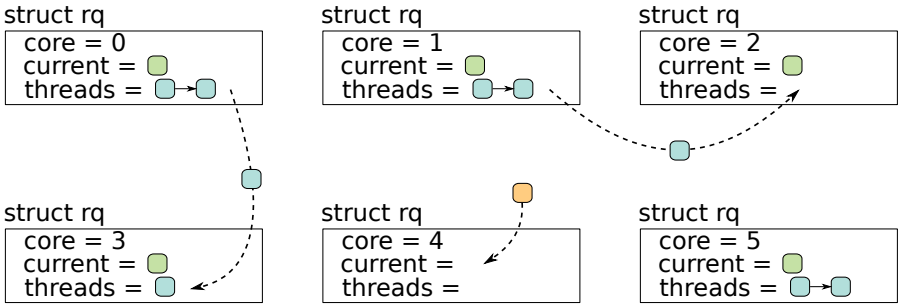


Figure 2.7: Distributed design of the Linux scheduler, with threads migrating to balance load (core 0 → 3 and 1 → 2), or waking up on idle cores (core 4).

thread terminated, e. g. normally or because of an exception. Threads in this state are colloquially called *zombies*.

Each thread also has policy-specific variables for each scheduling policy in Linux. For example, the real-time policy `SCHED_DEADLINE` stores a *deadline* and a *period*. On the other hand, the `SCHED_OTHER` policy computes a *load* that reflects the weight a thread has on the computing resources. These policy-specific variables will be discussed in more details in the sections describing each policy.

2.4.3 *Runqueues*

The Linux scheduler, just as the kernel in general, was originally designed for single-core machines. All subsystems were designed in a centralized way, with no concurrent accesses to scheduling data in mind.[12] When SMP support was introduced in 2006 with Linux v2.6, a Big Kernel Lock (BKL) was used for mutual exclusion. This solution scaled poorly and led kernel developers to move from this BKL to more fine-grained locking and designs that were able to scale with the number of cores.[13]

For the thread scheduler, this materialized by making it a distributed system: each core maintains a *runqueue* (a `struct rq`) that contains, among other things, the currently running thread and a set of runnable threads that are waiting for the CPU resource. Figure 2.7 shows this distributed design. Each core takes its scheduling decisions locally, as if it was operating on a single core machine. This reduces contention on locks, since each runqueue has its own lock and locking multiple runqueues is rarely needed. This distributed scheduler architecture also favors cache locality by design.

However, such a distributed design can lead to situations where work is unequally distributed among cores, as seen in Figure 2.7. To mitigate this, threads can be migrated by the scheduler from one core to another. This can happen because of particular thread-related events, e. g. thread creation or unblock after an I/O, or because of scheduler-triggered actions, e. g. periodic load balancing.

[12] On single core machines, disabling interrupts is enough to avoid concurrency problems between threads.

[13] This BKL removal process took 9 years to complete [16, 37].

2.4.4 *The Scheduling Class API*

The Linux kernel provides an internal application programming interface (API) that allows developers to implement a thread scheduler in the kernel: *scheduling classes*. It consists of a set of functions to implement, described in Table 2.1. The core code of the scheduler subsystem[14] does all the generic work that is common to all scheduling policies. This includes acquiring and releasing locks, changing thread state, enabling and disabling interrupts, ... Outside of this generic code, each policy has its own code that implements different scheduling algorithms.

[14] Defined in the kernel/sched/core.c file.

To illustrate the interactions between generic and policy-specific code, we describe the creation of a thread, be it with the `fork()` or `clone()` system call. First, the data structures representing the parent thread's state are copied to create the child thread. This includes the `struct task_struct`, the file descriptors, the page table address for multithreaded applications, etc... Attributes are then correctly initialized, e. g. the PID and PPID. Note that by default, a thread inherits its parent's scheduling policy. These operations are performed by the `copy_process()` function. When all is correctly set up, the thread scheduler is now able to make this new thread runnable.

The `wake_up_new_task()` function is responsible for making the newly created thread runnable. Let T be this newly created thread and S_T its associated scheduling class (e. g. `SCHED_OTHER`, `SCHED_FIFO`), here is the initial wakeup algorithm:

1. Lock T and disable interrupts.
2. Call the `select_task_rq()` function of S_T in order to choose on which core T will be enqueued. Let this core be C_{dst}.
3. Lock C_{dst}.
4. Call the `enqueue_task()` function of S_T in order to actually put T in the runqueue of C_{dst}.
5. Unlock C_{dst}, T and enable interrupts.

Only steps 2 and 4 involve policy-specific code. The rest is provided as is by the scheduler subsystem.

Providing this internal API allows developers to implement thread schedulers without reimplementing generic code and also helps minimizing the number of bugs. Indeed, developing in the Linux kernel is difficult, and it is hard to have a big picture understanding of all mechanisms involved. This lack of understanding could lead developers to do things improperly, such as incorrectly handling interrupts or locks.

Function	Description
`enqueue_task(rq, t)`	Add thread `t` to runqueue `rq`
`dequeue_task(rq, t)`	Remove thread `t` from runqueue `rq`
`yield_task(rq)`	Yield the currently running thread on the CPU of `rq`
`check_preempt_curr(rq, t)`	Check if the currently running thread of `rq` should be preempted by thread `t`
`pick_next_task(rq)`	Return the next thread that should run on `rq`
`put_prev_task(rq, t)`	Remove the currently running thread `t` from the CPU of `rq`
`set_next_task(rq, t)`	Set thread `t` as the currently running thread on the CPU of `rq`
`balance(rq)`	Run the load balancing algorithm for the CPU of `rq`
`select_task_rq(t)`	Choose a new CPU for the waking up/newly created thread `t`
`task_tick(rq)`	Called at every clock tick on the CPU of `rq` if the currently running thread is in this scheduling class
`task_fork(t)`	Called when thread `t` is created after a `fork()`/`clone()` system call
`task_dead(t)`	Called when thread `t` terminates

Table 2.1: Scheduling class API in the Linux v5.7 kernel. Only a subset of functions are presented and some function parameters were omitted for conciseness.

CAVEATS OF THIS INTERNAL API. Despite all its benefits, the scheduling class internal API still suffers some problems. First, the behavior of every function of the API is not strictly defined. For example, the `enqueue_task()` implementations currently available throughout the kernel can enqueue one, multiple or no thread in the runqueue. This means that in this function, anything can happen. This greatly limits the possibilities for static analysis tools or formalization in order to prove the correctness of the scheduler.

META-SCHEDULER. As stated previously, Linux complies with a large portion of the POSIX standards, including parts involving thread scheduling. To this end, it implements multiple scheduling policies that we describe in the following sections. However, this also means that if multiple policies have a runnable thread, a choice must be made by Linux to determine which policy has the highest priority. Linux chooses a simple fixed-priority list to determine this order. Figure 2.8 shows this priority list. When a thread is scheduled out, the scheduler subsystem will iterate over this list and call the `pick_next_task()` function of each class until a thread is returned.

Figure 2.8: Priority list of Linux's scheduling classes.

2.5 GENERAL-PURPOSE OPERATING SYSTEM SCHEDULERS

In this section, we present multiple general-purpose schedulers implemented in production OSs like Linux or Windows. We emphasize more on Linux since the contributions presented in this thesis are implemented on this OS. We present historical schedulers of Linux as well as the current one, CFS. We also present some competing schedulers that live outside the Linux mainline code.

2.5.1 *Former Linux Schedulers*

For a rather long time, Linux used a very simple scheduler, with the idea that thread scheduling was easy and not a problem.

> *Let's face it — the current scheduler has the same old basic structure that it did almost 10 years ago, and yes, it's not optimal, but there really aren't that many real-world loads where people really care. I'm sorry, but it's true.*
>
> *And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.*
>
> — Linus Torvalds, 2001 [173]

Reality proved to be a little bit different, and the Linux scheduler became a performance problem for many users. This led to multiple rewrites of the scheduler over the years. We now present the multiple schedulers that existed in Linux before the current one.

2.5.1.1 *Round-Robin Scheduler*

The first versions of Linux used a simple round-robin scheduler. Runnable threads are stored in a linked list and each thread is assigned a fixed time slice. When a thread completes its time slice or switches to a blocked state, the next thread in the list is elected. Unsurprisingly, this design works poorly in an interactive setup such as a desktop computer. Indeed, I/O-bound threads usually run very briefly and very frequently, i. e. they do not use their time slice, while CPU-bound threads use their whole time slice. This leads to unfairness between these two types of threads with a round-robin policy.

This information is extracted from the source code of earlier Linux versions. A full reconstruction of the `git` history of the Linux project is available at: `https://github.com/mpe/linux-fullhistory`.

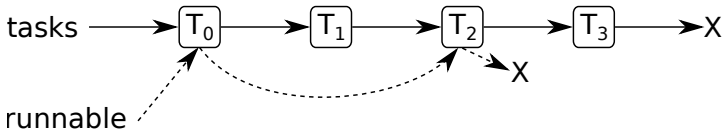


Figure 2.9: Data structures of the $O(n)$ scheduler in Linux.

2.5.1.2 *The $O(n)$ Scheduler*

To solve the unfairness between batch and interactive threads, the Linux community introduced the $O(n)$ scheduler in Linux v2.4 in 2001. The overall idea is that the scheduler assigns a time slice to each thread on the system. To do this, the scheduler divides time in *epochs*. At the end of each epoch, the scheduler reallocates a time slice to each thread and gives a bonus to threads that did not consume their whole time slice during the previous epoch.[15] All threads, runnable or not, are stored in a linked list, the `tasks` list. Runnable threads are also part of the `runnable` list, as shown in Figure 2.9. When the OS needs to choose a new thread to run, the $O(n)$ scheduler iterates through the `runnable` list and computes a *goodness* score for each thread. This goodness is computed with various metrics, such as the *nice* value or the time already consumed from its allocated time slice. The thread with the highest goodness is the one that will be given access to the core.

[15] The bonus is equal to half the time remaining in the thread's time slice.

Thanks to the goodness metric, this new scheduler was quite fair between threads, and interactivity was no longer penalized. However, the major caveat of this new scheduler is the algorithmic complexity of its operations. While the previous scheduler performed all operations in constant time, this new scheduler iterates through all runnable threads when choosing a new thread to run. This meant that the duration of the election depended on the number of runnable threads, hence the name of this scheduler, $O(n)$.

2.5.1.3 *The $O(1)$ Scheduler*

With the advent of multi-core architectures and multithreaded programming, the complexity of the $O(n)$ scheduler became prohibitive. The $O(1)$ scheduler, as well as proper support for SMP architectures, was introduced in 2003 with Linux v2.6 to solve this problem. This description is derived from code reading and from the excellent *Linux Kernel Development (2^{nd} Edition)* book from Robert Love [111].

As shown by Figure 2.10, a runqueue consists of two *priority arrays*, i. e. arrays of lists, with 140 entries each: the *active* and *expired* arrays. Each entry corresponds to a different priority level: 0 to 99 are real-time priorities and 100 to 139 are normal priorities.[16] When a new thread must be elected by the scheduler, the highest priority thread in the *active* array is selected. In our example, T_0 would be the first thread to be elected. When a thread uses all its time slice, it is moved

[16] These 40 levels of normal priorities are mapped to the nice value that ranges from -20 to 19.

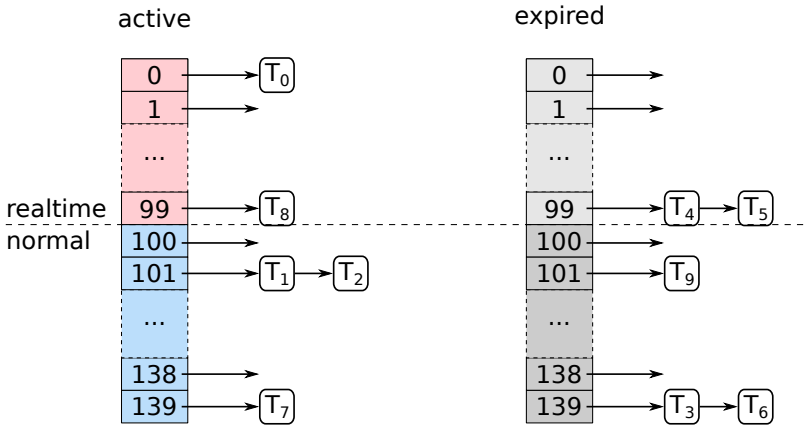


Figure 2.10: Data structures of the $O(1)$ scheduler in Linux.

Nice	Time slice
+19	5 ms
0	100 ms
-20	800 ms

Table 2.2: Time slice allocations in $O(1)$.

from the *active* to the *expired* priority array, its time slice is recomputed, and another thread is scheduled. When the *active* array is empty, both arrays are swapped: *expired* becomes *active* and vice versa. The time slice allocated to a thread depends on its priority, as shown by Table 2.2. This new design allows for good balance between interactive and batch threads, thanks to priorities, and does so while maintaining a good $O(1)$ algorithmic complexity, thanks to priority arrays.

SMP SUPPORT. The SMP support in the scheduler consists of moving from a centralized runqueue to distributed runqueues, each core scheduling threads locally without knowledge of other cores. This distributed scheduler architecture brings new challenges. From now on, since each core only schedules the threads in its runqueue, threads can be asymmetrically distributed among cores. One core could have 10 threads to schedule while another one could have only one single thread to schedule, leading to unfairness between threads depending on the core they are placed on. To solve this problem, the scheduler performs load balancing by migrating threads between cores in order to even the number of threads of all cores. This balancing is performed on two occasions: *(i)* when a core becomes idle, i. e. no runnable thread is locally available, *(ii)* periodically, every millisecond when the system is idle, every 200 ms otherwise. Load balancing is done with a *work stealing* approach: each core does its own balancing and tries to steal threads from the busiest core on the system. To reduce the cost of this balancing and the number of migrations, balancing has some constraints: threads are migrated only if the imbalance between cores exceeds 25%, and cache-hot threads, i. e. threads in the *active* array, are less likely to be migrated than other threads, i. e. threads in the *expired* array.

2.5.2 *Completely Fair Scheduler*

The Completely Fair Scheduler (CFS) is the default scheduler in Linux since 2007, with version 2.6.23. The general idea behind this scheduler is to *"model an ideal, precise multi-tasking CPU on real hardware"* [29]. The meaning behind this is that CFS aims at emulating an ideal CPU that would execute all runnable threads in parallel, allocating each thread the exact same computing power simultaneously. For example, if 4 threads are runnable, each thread should be allocated 25% of the CPU power at all times. However, since real hardware provides a finite number of cores that can simultaneously run one thread each, CFS has to model such an ideal hardware. We will first describe how CFS manages the allocation of CPU time to threads, and then detail the thread placement strategy of CFS. Most of the information of this section is extracted from the Linux v5.4 kernel source code and from Robert Love's excellent book, *Linux Kernel Development, Third Edition* [112].

2.5.2.1 *Election in CFS*

As described previously, Linux opts for a distributed scheduler design, where each core is responsible for scheduling its threads, and balancing happens periodically or because of events such as a thread waking up or a core becoming idle. CFS applies its general idea to each core separately, and each core is "split" between its threads. Each thread is assigned a time slice[17] that depends on the number of threads present in the core's runqueue and the thread's *weight*. This *weight* depends directly on the nice value of the thread.[18] The higher the *weight*, the longer the allocated time slice. This way, each thread in the runqueue of a core gets assigned a portion of the CPU time this core offers, thus emulating an ideal CPU, while still maintaining a form of priority-based scheduling.

[17] Also called a quantum.

[18] This nice-to-weight mapping is hardcoded in the kernel. A high weight means a low nice value and a high priority.

When the CFS scheduling class is asked for a thread to schedule, the currently running one is scheduled out, and CFS chooses the thread that needs the CPU the most to strive towards an ideal CPU. To this end, CFS introduces the notion of *vruntime*. The *vruntime* of a thread represents the time a thread has executed, in nanoseconds, adjusted by its *weight*. For the same real execution time, the *vruntime* of a thread with a small *weight* will increase faster than the one of a thread with a large *weight*. When election time comes on a given core, CFS will choose the thread with the lowest *vruntime* on this core, and the scheduled out thread's *vruntime* is updated to reflect its last usage of the CPU and the thread is put back in the runqueue. However, these operations can be costly with the wrong data structure. With CFS, threads are stored in a red-black tree [73] ordered by *vruntime*. The leftmost thread in the tree is the one with the lowest *vruntime*, and scheduled out threads are inserted in their correct position to keep

the structure sorted. All these operations are performed efficiently, in $O(\log n)$ complexity.

Additionally, since CFS is a *preemptive* scheduler, the election procedure is not only called because the currently running thread relinquishes its core, i. e. calls the `yield()` system call. A thread can be preempted if it used up all its time slice. This condition is checked at every tick. Another preemption reason is the waking up of a thread with a higher priority than the currently running one.

2.5.2.2 *Thread Placement in CFS*

The decentralized architecture of the Linux scheduler can lead to performance below basic expectations. Indeed, a situation where a core has 4 threads in its runqueue while all other cores have none is a waste of resources. This introduces the need to balance threads between cores. To do so, CFS introduces the notion of *load* used to represent how much a thread uses a core.

LOAD. The **load** of a thread corresponds to how much time it was runnable compared to the time it could have been runnable, weighted by its *weight*. Therefore, a thread that spends half its time sleeping will have a load of 50%, no matter how long it actually ran. However, the behavior of a thread can change over time. Because of this, the more time passes, the less this cumulative *load* has meaning.

The notion of **average load** is therefore introduced to account for the current and past *load* values differently [38]. Time is divided into 1 millisecond periods, and *load* is computed for each period. The *average load* L_{avg} is based on the *load* of the current period L, the previous *average load* and a constant y:

$$L_{avg} = L + y \times L_{avg}$$

With $y < 1$, the current *load* has more impact on the *average load* than older *load* values that decay every millisecond. The value of y has been chosen such that $y^{32} = 0.5$, meaning that the weight of a given load is halved every 32 millisecond in the *average load*.[19] When the CPU is not used by a process during a period, the *average load* therefore decreases.

[19] An informed reader might wonder how this is done in kernel space where floating operations are frowned upon: large integers are used to approximate these computations.

In order to decide if threads must be migrated from one core to another, CFS can compute the imbalance between cores with these metrics. CFS triggers its migration mechanisms on two types of occasions:

- Thread-related scheduling events: thread creation (`fork()` or `clone()`), program replacement (`exec()`), thread unblocking (transition from `(UN)INTERRUPTIBLE` to `RUNNING`),
- Core-related scheduling events: when a core becomes idle, i. e. no more runnable thread available, or periodically.

The first type only places the thread concerned by the event, while the second can migrate a bulk of threads at once. Additionally, each one of these occasions perform balancing differently, especially regarding the hardware topology of the machine (see the *Hardware topology* insert for more information).

Hardware Topology

During the boot of the kernel, Linux queries the hardware to fetch hardware information, such as which cores share SMT capabilities or caches, as well as the topology of the NUMA interconnect. From this information, Linux builds a model of the topology in the form of *scheduling domains*. Figure 2.11 shows the scheduling domains built for a 4-socket NUMA machine. Each domain is tagged by its hardware particularity: `SMT` means that cores in this domain share SMT capabilities, `LLC` means that cores share an last level cache and `NUMA` means that memory access times may not be uniform in this domain. This topological information is used to change the minimal imbalance needed for periodic load balancing:

- 10% imbalance for an `SMT` domain,
- 17% for an `LLC` domain,
- 25% for `NUMA` domains.

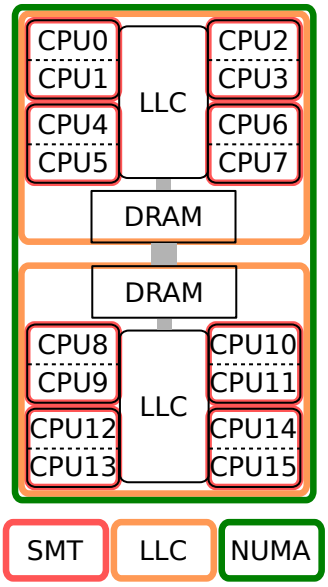


Figure 2.11: Scheduling domains of a 4-socket NUMA machine.

THREAD PLACEMENT. Thread creation, i. e. `fork()` and `clone()` system calls, is an ideal time to migrate a thread. Indeed, the created thread most likely won't share much data with its parent, meaning that ensuring cache sharing between these two threads is unimportant. This means that this newly created thread can be placed on any core of the machine. In practice, CFS will try to find the idlest CPU in terms of load on the whole machine.[20] Program replacement, i. e. `exec()` system call family, is also a great opportunity for migration since the thread will change its code and data, and therefore does not need any of its previously loaded cache lines. Placement behavior is the same as for thread creation.

When a thread goes from a blocked state to a runnable state, i. e. `(UN)INTERRUPTIBLE` → `RUNNING`, CFS must determine on which core the thread should be enqueued. Since the thread has run, and has probably fetched code and data in the caches of the core previously used, it is beneficial to try to keep it close to its previous core. CFS will search for an idle core in the `LLC` domain. If one is found, the thread will be placed on this idle core. Else, the previous or waker core will be selected, depending on which will be the most quickly available.

[20] On multi-hop NUMA machines, the search will not go beyond a certain number of hardware-defined hops.

IDLE BALANCING. Another good opportunity for thread balancing is when a core becomes idle. Indeed, a core becoming idle has no thread to run, and can therefore perform more heavyweight work to balance cores on the machine. When a call to `schedule()` returns no runnable thread, CFS performs **idle balancing**, which consists in stealing work from other cores and placing it on this newly idle core. The search for stealable threads is performed in each scheduling domain where the newly idle core is present, from the smallest to the biggest. As soon as the newly idle core has threads to run, it will stop stealing threads and perform a new election. Note that threads can be stolen in bulk in order to even out the imbalance between cores.

LOAD BALANCING. Finally, as a last safety net, CFS periodically performs **load balancing** in all scheduling domains. Unlike idle balancing, periodic load balancing does not stop when all cores are not idle. Its goal is to minimize the imbalance between all scheduling domains, whatever the level of the domain. Since this operation can be costly and not very scalable, periodic load balancing is performed concurrently in each domain, at different times.

Balancing a domain means evening out the average load of each of its child domains. For example, balancing an `LLC` domain D_{LLC} means that all `SMT` domains within D_{LLC} have a similar average load. The period between each balancing of a domain depends on the number of cores in the domain: a domain with N cores will be balanced every N milliseconds. This period can be extended if consecutive balancing fail, i. e. the load is already balanced and no thread migration is performed. The period is capped to a maximal value of $2N$ and reset when a balancing succeeds.[21]

[21] The maximal value can be modified at run time in the *procfs*.

2.5.2.3 *Other Miscellaneous Features*

In addition to the general features of CFS presented earlier, there are other smaller features with the sole objective of optimizing specific patterns. Here is a description of two of these features, presented as an example.

GROUP SCHEDULING. CFS features **group scheduling** to extend fairness. The default behavior of CFS is to ensure fairness between all threads. However, on a multi-user system, a situation where user A runs 2 threads and user B runs 98 threads is going to be detrimental to user A that will be given only 2% of CPU time. To solve this issue, *group scheduling* considers groups of threads when allocating CPU time. This is done at the user level, i. e. each user gets an equal share of the CPU, and this share is then divided among the user's threads, but also at the process, *TTY* and *cgroup* levels.

CACHE NICE TRIES. Another optimization aimed at improving cache locality is **cache nice tries**. In order to prevent cache-hot threads from being migrated easily, the scheduler can migrate these threads only after failing to balance multiple times consecutively. For example, to migrate a cache-hot thread from a NUMA node to another, it will take two consecutive failures before being able to do the migration.

2.5.3 *Other Scheduling Policies in Linux*

The Linux kernel features other scheduling policies in addition to CFS. Some are defined by the POSIX standard, i. e. `SCHED_FIFO` and `SCHED_RR`, while others are specific to Linux, i. e. `SCHED_DEADLINE`, `SCHED_BATCH` and `SCHED_IDLE` [115].

If you recall Figure 2.8, you will notice that there are more policies than scheduling classes. Indeed, some classes implement multiple policies. The `rt` scheduling class contains both the `SCHED_FIFO` and `SCHED_RR` policies, while the `fair` class implements `SCHED_BATCH` and `SCHED_IDLE` in addition to CFS. Note that the `idle` scheduling class is different from the `SCHED_IDLE` policy. It is only used by a special thread, called the *idle* thread, that is executed when nothing else is runnable. Its job is to activate architecture-specific features to lower the energy usage of the CPU.

SCHED_FIFO. The `SCHED_FIFO` policy implements a priority-based First-In First-Out (FIFO) scheduler. When a thread must be chosen to run, the first thread from the highest priority FIFO list is selected. If a higher priority thread becomes available, it preempts the currently running one. There is also no concept of time slice, threads relinquish the CPU by yielding it, blocking or if a higher priority thread wakes up. In the latter case, the thread is not moved to the end of its list, while in the other cases, it will be.

SCHED_RR. The `SCHED_RR` policy implements a priority-based algorithm similar to $O(1)$, with one list per priority level. It is actually exactly the same code as `SCHED_FIFO`, with the exception of the addition of time slices. The selection is performed in a round-robin fashion, with fixed time slices that can be configured through the `procfs`.[22] When a thread is preempted by a higher priority thread, the remaining of its time slice stays untouched, the thread is allowed to finish it when scheduled again.

[22] The default value is 100 ms.

SCHED_DEADLINE. This policy implements a deadline real-time scheduler based on global Earliest Deadline First (EDF) [110] and Constant Bandwidth Server (CBS) [2]. This category of schedulers is usually used for real-time periodic tasks. When a thread uses this policy, three values must be specified by the user: a *runtime*, a *deadline*

and a *period*. The *runtime* corresponds to the estimated completion time of the task.[23] The *deadline* is the time before which the task should complete, relative to its start time. For each window of time of a duration equal to *period*, the task should be run once. Naturally, these values must be chosen such as $runtime \leq deadline \leq period$. The policy then computes the order of execution of all its threads in order to fulfill all deadlines. For this to be possible, the scheduler performs an admittance test when a new thread wants to use this policy.

[23] For hard real-time tasks, this value usually corresponds to the Worst-Case Execution Time (WCET) [181].

A NOTE ON STARVATION. The three previously presented strategies are considered *real-time* policies. A misuse of one of these policies can cause a starvation for all threads with a lower priority on the system, thus effectively freezing the machine. For example, a high priority infinite loop, may it be due to a bug or a malign attack, would hog the CPU and be hard to kill. To avoid this, the scheduler subsystem provides a safety net with two configuration variables that can be tweaked through the `procfs`: `sched_rt_period_us` and `sched_rt_runtime_us`. The ratio between the latter and the former gives the maximum percentage of time allocated to real-time threads. The remaining is given to "normal" threads, giving them a chance to act and fix the state of the system if necessary.

SCHED_BATCH. This policy uses the same algorithm as CFS, except that it will consider threads as CPU-intensive. This means that a small penalty will be applied in terms of interactivity, i. e. threads waking up will not be placed in a favorable position in the red-black tree. However, the allocated time slice will be longer, thus reducing the number of preemptions due to an expired quantum.

SCHED_IDLE. This policy is used for very low priority tasks that should not disrupt the execution of any other thread. The threads in this policy will be scheduled only if no other thread is runnable. This is useful for users with long running background tasks and interactive tasks running together, e. g. a compilation and a web browser.

2.5.4 *Brain Fuck Scheduler and Multiple Queue Skiplist Scheduler*

Although CFS tries to maximize performance for all types of applications and hardware, it is a difficult task to achieve. Therefore, there exists other schedulers for the Linux kernel that target more specific use cases. The most famous one is the Brain Fuck Scheduler (BFS), developed by Con Kolivas,[24], and its successor, the Multiple Queue Skiplist Scheduler (MuQSS). Kolivas developed these schedulers with one single target in mind: desktop machines with a limited number of cores. The focus was put on maximizing interactivity and simplicity in its design.

[24] Kolivas is a kernel hacker that largely contributed in the development of schedulers in Linux (he was credited by Ingo Molnar as an inspiration for CFS) although his actual profession is anesthesiologist.

2.5.4.1 *Brain Fuck Scheduler*

The Brain Fuck Scheduler (BFS) [96], first released in 2009, is an alternative to CFS aiming at using a simple algorithm, devoid of tunable parameters, that performs well on desktop machines with up to 16 cores. Threads are stored in a global priority array.[25] When election time comes, if a real-time priority thread is available, the one with the highest priority is selected. If none is available, Brain Fuck Scheduler (BFS) iterates through all runnable threads on the system and picks the one with the *earliest deadline*. The deadline of a thread is derived from its priority and the last time the thread used up its time slice. All threads are allocated the same fixed time slice of 6 ms.[26]

[25] There is no expired array as in $O(1)$.

[26] This time slice value is the only tunable of BFS, with a file in the *sysfs*.

Having a centralized runqueue and a deadline-centered approach is supposed to improve the overall performance of desktop systems that mostly run interactive applications. In 2013, a user benchmarked and compared BFS with CFS on 7 different machines with 1 to 16 cores [69]. The benchmarks focused on three workloads: compilation, compression and video encoding. Overall, both schedulers perform equally, the largest difference in performance being 8% in favor of BFS. CFS developers also compared their work with BFS, noticing no performance difference [125]. This shows that a simpler scheduler does not equate a worse scheduler. In addition, BFS is used in production Linux distributions such as GalliumOS [62], PCLinuxOS [136] or Zenwalk [191].

2.5.4.2 *Multiple Queue Skiplist Scheduler*

With desktop machines having more and more cores, having a single global runqueue for all cores became a bottleneck for BFS, due to the contention on the runqueue lock. To work past this contention issue, Kolivas designed an evolution of BFS: the Multiple Queue Skiplist Scheduler (MuQSS) [97]. The idea is to keep the general ideas behind BFS, i. e. earliest deadline first election and no load balancing, with scalable data structures for modern multi-core hardware. The single global runqueue design was changed to a per-core runqueue design in order to reduce the complexity of the lookup operation on the previous implementation. The linked list data structure is also replaced by priority-ordered skip lists [143]. When a core needs a new thread to schedule, it will check the first thread of each runqueue, i. e. the thread that most likely would have the earliest deadline[27] without holding any lock, and select the thread with the earliest deadline among these threads.

[27] The definition of a thread's deadline is similar to the one used in BFS.

Hardware topology awareness is also accounted for in MuQSS thanks to thread placement at wakeup time, as well as some optimizations in the election process. When a thread wakes up, the choice of its future runqueue will depend on multiple factors such as core idleness, cache hotness or the number of threads already in the runqueue.

Additionally, at election time, when runqueues are scanned, a higher priority will be given to idle cores or cores sharing caches. Balancing is only done during these two occasions, there is no active balancing algorithm like in CFS.

2.5.5 *FreeBSD's ULE Scheduler*

FreeBSD is a general-purpose UNIX-like OS. It is the most widespread distribution of the Berkeley Software Distribution (BSD) family. Until 2003, FreeBSD implemented a modified version of the historical 4.3BSD scheduler [118]. This scheduler is a time-sharing priority-based scheduler. Priorities are computed with an estimation of the recent CPU usage and the *nice* value of the thread. Fairness is ensured only among threads with the same priority, with a round robin election scheme and fixed time slices of 100 ms. FreeBSD added the support for SMP architectures, as well as a real-time scheduling class. Due to its algorithmic complexity, this scheduler does not scale with a large number of threads.

In 2003, the new ULE scheduler [149] was introduced in FreeBSD.[28] ULE heavily relies on the distinction between I/O- bound and CPU-bound applications. ULE determines the *interactivity* of each thread based on the time spent running and the time spent voluntarily sleeping during the last five seconds. This score, in addition to the *nice* value of the thread, is used to compute the thread's *priority* and classify threads as *interactive*, *batch* or *idle*.

[28] The name ULE comes from the last three letters of *schedule* and was the username used for testing purposes by the developer.

With this classification in mind, each core has three runqueues sorted by *priority*, one for *interactive* threads, one for *batch* threads and one for *idle* threads. The *interactive* runqueue has one FIFO list per priority, in a similar fashion to the priority arrays of the $O(1)$ scheduler in Linux (see Section 2.5.1.3). The *batch* runqueue, on the other hand, is sorted by runtime, weighted by the nice value, in a similar fashion to CFS (see Section 2.5.2.1). The *idle* runqueue, finally, contains threads that want to run only if nothing else is available, among them the *idle* thread.

When choosing the next thread to schedule, each runqueue is inspected in the aforementioned order until a thread is found. Interactive threads have absolute priority over batch threads, which means that batch threads may face starvation. However, developers see this as a minor problem since interactive threads sleep more than they run, leaving time for batch threads to run.

When scheduled, a thread can execute for a given time slice that does not depend on its priority, but only on the number of threads on its core. A budget of 10 ticks is equally shared among threads, with each thread getting a minimum of one tick to execute. At every clock tick, ULE checks if the allocated time slice has been used and forces a

preemption if it is the case. Preemption can also happen if a higher priority thread wakes up.

In terms of load balancing, ULE only tries to even out the number of threads per core. When placing a newly created or unblocking thread, ULE behaves in a similar manner to CFS. It tries to place the thread close topologically to its previous or its parent's core. In addition, ULE balances the number of threads per core periodically. Unlike CFS, only one thread is migrated at a time.

RELATED PUBLICATION. We published a comparison of CFS and ULE to determine which one was *the best* [23]. We did this by implementing ULE in Linux. The results showed that neither scheduler was better than the other. Depending on the workload, the winner was different. This shows that there is no silver bullet in terms of general-purpose scheduling. The detailed results of this publication will not be presented in this thesis.

2.5.6 *Windows Scheduler*

The Windows OS provides some information about its thread scheduler in its online documentation [123]. Unfortunately, since the source code is not available, we are not able to give more precise information or check that the provided information is valid. The general ideas of the Windows scheduler are the same since at least Windows XP, released in 2001.

Windows implements a priority-based scheduling algorithm with priorities ranging from 0 to 31. It uses the same kind of algorithm as ULE for *interactive* threads. All threads with the same priority are scheduled in a round robin fashion. When choosing a thread, the scheduler checks all priority levels, from highest to lowest priority, until a thread is found. When a thread with a higher priority than the one running becomes available, preemption is triggered.

The *priority* of a thread is computed with two criteria: its *priority class* and its *priority level* within the class. There are 6 priority classes containing 7 priority levels each.[29] For example, by default, a thread is in the `NORMAL` class with the `NORMAL` level, which corresponds to a priority of 8. Threads are able to choose their priority class and level by themselves, which means that the Windows scheduler relies on developers to be responsible when choosing the priority they wish to use.

[29] Multiple pairs of class and level can map to the same priority, hence the difference between the 42 combinations and the 32 priority values.

Regarding thread placement on SMP systems, it is not clear from the documentation whether there is a single shared data structure containing all threads or if threads are distributed like in CFS or ULE. The only information stated is that threads can run on all cores, unless developers decide to use a subset of cores by setting a specific *thread affinity*. Additionally, threads can specify an *ideal processor* to the

scheduler. This value does not guarantee that this core will be used for the thread, it is merely a hint given to the scheduler.

Regarding NUMA architectures, the Windows scheduler tries to place threads close to the memory they are using. Memory is allocated on the local node by default, so the thread should stay on the same node during its execution. However, if memory must be allocated on a remote node, e. g. the local memory is full, the thread will run on another node if it uses memory from that said node.

In addition to its kernel scheduler, Windows offers a particular feature since Windows 7, User-Mode Scheduling (UMS) [124]. The idea is to allow developers to schedule the threads of their application by themselves, without relying on the kernel at all. The application embeds its own thread scheduler and is able to perform context switches without going through the kernel. The only interaction between the kernel scheduler and the UMS happens when a thread wakes up: the kernel scheduler notifies the UMS of this event and lets it process it. In this context, the kernel scheduler's job is only to allocate CPU resources to the process, and the UMS will take the scheduling decisions within the process.

2.6 USER-LEVEL SCHEDULERS

In addition to thread scheduling at the OS level, there exists multiple ways to perform thread scheduling at the user level. Some programming languages offer threading by design while others offer threading through the use of specific libraries. Regardless of the way threading is offered, it gives an opportunity to perform scheduling. This can be implemented with different thread mapping models that define the interaction between user level and kernel level threads. In this section, we present these thread mapping models and detail a few examples of user level schedulers.

2.6.1 *Thread Mapping Models*

When implementing threading in a runtime system (language or library), the interaction with the underlying OS threads is a key design choice. In the following, we will distinguish both types of threads: threads managed by the OS will be called *kernel threads* while threads managed by the runtime system will be called *user threads*.

The easiest and most common mapping model is the **1:1 mapping**, where each user thread is mapped to a kernel thread. In this model, the threading system completely relies on the OS to manage threads. All scheduling decisions are taken at the OS level. This model requires OS support in order to be able to create threads.

Another model is the **N:1 mapping** where all user threads are mapped to a single kernel thread. In this model, the OS has no knowl-

edge of the existence of user threads. The runtime system must manage its threads and perform context switches. This approach can be used on OSs that do not support the creation of threads. Another possible advantage is the reduced cost of context switching in user space, without the need to go back and forth into kernel space [8, 166].[30] The major limitation of this model is that a single user thread is able to run at a time. This does not allow to exploit processors featuring SMT or SMP.

[30] This argument is questionable since no recent study shows this difference on modern processors or modern OSs.

The **M:N mapping** maps multiple user threads (M) to multiple kernel threads (N). This approach is a combination of the two previous models. The goal is to let the runtime system manage its threads while taking advantage of multiple kernel threads, allowing multiple threads to run simultaneously. Such systems must implement a scheduler that decides which user thread runs on which kernel thread. This model is more complex than the previous ones because it contains two schedulers with limited means of communication. This can lead to subpar performance when both schedulers do not coordinate correctly in their respective decision making. This hybrid model is extensively used in runtime systems at the language or library level.

In runtime systems that do not fully rely on the OS for thread management, user threads are sometimes called **green threads**. One general caveat of green threads is the handling of blocking operations. When a blocking operation is performed by a green thread, the underlying kernel thread is blocked, effectively blocking all other green threads sharing it. This can be solved by the use of more complex asynchronous I/O operations and wait-free algorithms.

Fibers are another type of thread that fully rely on cooperative multi-threading instead of preemptive multi-threading. The concept of fibers is similar to **coroutines** in programming languages theory. Fibers can be implemented in user space with minimal OS support, with an N:1 or N:M threading model.

Note that for user level schedulers to be efficient, support from the OS is necessary. The user scheduler needs to be notified by the kernel of any scheduler-related event that might influence scheduling decisions. This was brought to light by Anderson *et al.* [8] in their work on scheduler activations.

2.6.2 *Language Runtime Systems*

With the different threading models in mind, we present some language runtime systems that provide threading natively. These runtime systems range from fully fledged VMs to language standard libraries. We mainly focus on N:M model implementations since they usually provide more complex schedulers.

JAVA. The Java VM specification does not specify how threading should be implemented, it is implementation-defined. Most Java VMs, including HotSpot, the *"official"* VM maintained by Oracle, use a 1:1 threading model [54, 77, 87, 88, 99, 169]. The discontinued Java VM JRockit [131] provides both a 1:1 mapping and an N:M mapping when using **Thin Threads**. Unfortunately, no details on the scheduling algorithm are provided in the documentation.

HASKELL. The Glasgow Haskell Compiler (GHC) provides a runtime system that implements an N:M threading model [116]. The runtime creates one kernel thread per core, and schedules its green threads on them. Green threads sharing a kernel thread are scheduled in a round robin fashion. The GHC runtime also provides load balancing mechanisms: green threads are not indefinitely bound to the same kernel thread. When a GHC runqueue contains more than one thread and other runqueues are empty, green threads are migrated to avoid having idle cores. Additionally, the GHC runtime does not migrate threads on wake up to favor cache locality.

GO. The Go runtime system implements an N:M threading model [93, 177]. When a Go application starts, the runtime creates one kernel thread per core and assigns an execution environment to each one. When the program starts a *goroutine*,[31] it is assigned to an execution environment. Each execution environments features a cooperative FIFO scheduler for its goroutines. Context switches can be triggered by the program with the `go` keyword, when executing a system call, when using a synchronization primitive or when the garbage collector is executing. When an environment has nothing to run, it tries to steal goroutines from other environments on the system. Goroutines can also be migrated to avoid blocked kernel threads. Ongoing work on the Go runtime aims at implementing preemptive scheduling [34, 35].

[31] A goroutine is a green thread in Go dialect.

OTHER LANGUAGES. Many programming languages provide a runtime system with threading models similar to what we already presented. Erlang [63], Dyalog APL [52] natively support green threads with an N:M model. Python [170], Racket [171], PHP [140], Lua [114], Tcl [167], Julia [89] support an N:1 threading model, usually by supporting coroutines. Note that some of these languages provide an API to create kernel threads, enabling developers to implement an N:M threading model. Some languages are specifically designed for parallel computing, usually targeting High Performance Computing (HPC) applications on supercomputers, such as X10 [30], Chapel [27] or Fortress [102]. Fibers are natively supported by languages like Crystal [41], or even through an OS interface in Windows [121]. The UMS feature of Windows presented in Section 2.5.6 is also a form of N:M mapping provided by the OS.

2.6.3 *Threading Libraries*

In addition to runtime systems natively provided by languages, external libraries also implement such threading models. These libraries use different models and interface themselves in different ways with programming languages. Some act as normal libraries and provide support for coroutines. Others act as language extensions used to express parallelism and generate multi-threading at compile time.

COROUTINE LIBRARIES. Most libraries provide support for coroutines in an N:M threading model. Python has the `greenlet` [70] and `stackless` [172] libraries among others. Kotlin provides coroutines through a first-party library [25]. In C, the GNU Portable Threads [55], State Threads [156] and Protothreads [51] libraries implement coroutines in a N:1 model. These libraries usually use a simple round robin algorithm to select which coroutine should run next.

LANGUAGE EXTENSIONS. Cilk [61] extends C and C++ with new constructs that express parallelism. For example, the `spawn` keyword in front of a function call causes this function to be run in another thread asynchronously. Cilk also features loop parallelization and vectorization of array operations. Threading is managed with an N:M model, with a work stealing strategy to even the load of each underlying kernel thread. OpenMP [44] provides pragmas for C, C++ and Fortran to express parallelism. These pragmas are used to tag loops or code blocks as candidates for parallelization. The runtime environment then distributes work between multiple kernel threads. The behavior of the runtime can be customized with environment variables in order to decide how threads are moved between cores.

Some work on these systems was also proposed to enhance their resource management. Callisto [75] and SCAF [40] extend OpenMP with a scheduling layer that manages resources across concurrent jobs. This approach allows these systems to reduce the interference between OpenMP applications and improve overall performance.

2.7 HYPERVISOR SCHEDULERS

Hypervisors can be classified into two categories: **type-1** hypervisors that run directly on hardware and **type-2** hypervisors that run on top of an OS [139]. Type-2 hypervisor do not need to implement a scheduler since they can rely on the underlying OS to do this job. Type-1 hypervisor, however, must implement a scheduler since it is a feature needed to use the computing resources, i. e. cores.

2.7.1 *Production Hypervisors*

TYPE-1. VMware ESXi [168], Xen Credit [185] and Credit2 [186] and Microsoft Hyper-V [122] hypervisors implement proportional fair share algorithms and load balancing similar to what is implemented in CFS. Xen also provides a real-time scheduler with RTDS [187]. Oracle VM Server [165] implements a priority-based scheduler. Nutanix AHV [130] relies on Qemu/KVM for the virtualization part, and only manages the placement of vCPUs on cores. It tries to optimize performance by minimizing contention on storage and network devices.

TYPE-2. The QEMU/KVM hypervisor creates one thread per vCPU and lets Linux do the scheduling [176]. VirtualBox [132], Parallels [133], bhyve [179] provide no information in their documentations about scheduling. We therefore assume that they do nothing in terms of scheduling and rely on the underlying OS. This is not a surprise to see this trend in type-2 hypervisors because scheduling is difficult, not necessary to implement in this setup and might clash with the existing OS scheduler. A project from Samsung, CFS-v [158], modifies the Linux scheduler and QEMU/KVM to improve the I/O performance of VMs at the expense of some computing performance.

2.7.2 *Research Prototypes*

Researchers mainly focus on type-1 hypervisors, usually in the context of cloud computing. In this context, the main problem is to respect VMs' applications requirements, such as tail latencies, and maximize resource utilization. Indeed, a simple way to improve the performance of applications with regards to latency-related metrics is to overprovision the VMs. This solution is adequate from the perspective of the client, but induces a waste of resources, and therefore money, for the cloud provider.

Tableau [175] uses real-time techniques to guarantee minimal CPU usage and maximal bounds on scheduling delays. RTVirt [193] proposes cross-layer scheduling where the guest and the host cooperate to improve the performance of time-sensitive applications. Other systems flatten the scheduling hierarchy: guests communicate information to the host, and the host performs all scheduling operations [49, 103]. Conversely, recent work leverage the hierarchy of schedulers for real-time tasks, arguing that this is a more realistic and applicable approach [1].

2.8 CONCLUSION

In this chapter, we presented a wide range of schedulers that try to achieve different goals. The scheduler code rapidly becomes large and complex, as more features are implemented. These features can be needed to handle a new hardware characteristic or to respect a given scheduling property. From what we observed and learned, we identify three axes of improvement: **scheduler development**, **performance enhancement** and **application-specific schedulers**.

SCHEDULER DEVELOPMENT. Developing a scheduler is a difficult task that requires knowledge of scheduling, hardware and low-level kernel programming. This complexity increases the likelihood of producing incorrect code. The incorrectness can be located in the scheduling algorithm itself, e. g. one could think that the algorithm is work-conserving while it is actually not. It can also be located in the implementation of the scheduler and cause crashes or undefined behaviors at run time.

This first axis aims at easing the development of new schedulers by alleviating the risk of making mistakes. We can do this by providing a complete tool chain containing a DSL and its compiler. From this high level abstract language, the compiler could generate efficient low-level C code for Linux. In addition, we could also add a verifier to our tool chain to formally verify that some scheduling properties cannot be violated. This axis is treated in Chapter 3.

PERFORMANCE ENHANCEMENT. In addition to the safety bugs previously mentioned, schedulers are also highly subject to what we could call *performance bugs*. They do not cause actual crashes, but they silently eat away at performance. This makes it very difficult to notice. The only way of noticing them is to produce another scheduler that performs better or use profiling tools that highlight the problem.

The second axis aims at providing profiling and visualization tools that enable scheduler developers to detect performance bugs, identify the source of the problem and direct them toward a solution. This axis is treated in Chapter 4, with a highlight on a performance bug related to dynamic frequency scaling on modern processors.

APPLICATION-SPECIFIC SCHEDULERS. One implication of the two problems we will address in the first two axes is the difficulty to develop schedulers. Producing a correct scheduler that performs well is a challenge that discourages most developers. Because of this, there is a very limited number of schedulers developed, and the ones in general-purpose OSs aim at being generic, thus becoming extremely large and complicated, e. g. CFS. It also becomes difficult to evaluate the impact of specific features on a specific workload since most

features are intertwined. We cannot know which feature is beneficial and which is detrimental to the performance of an application.

The third axis aims at helping end users choose the best possible scheduler for their application. We propose a feature-based model of a scheduler that allows each feature to be evaluated individually and multiple methodologies to find the best combination of features for a specific application. This axis is treated in Chapter 5.

3

WRITING SCHEDULERS WITH IPANEMA

The scheduler is a core component of the operating system (OS). Thread ordering and placement performed by the scheduler highly influence the **performance of applications** on the system. These decisions are also vital to efficiently use hardware resources. The scheduler code itself can also be a source of performance. For example, Google data centers spend up to 6% of all CPU cycles executing scheduler code [90].

However, **writing a scheduler is a daunting task**. Indeed, a poorly designed scheduler can have a negative impact on performance and provoke crashes or cause the system to hang. It also requires good low-level programming skills since schedulers are implemented in kernel space. These skills should also include debugging in the kernel, since it is a non-trivial operation that requires knowledge of the OS and of the debugging tools associated.[32] These difficulties limit the development of new schedulers and incite developers to implement a single generic scheduler like CFS in Linux.

From these two observations, even if writing specific schedulers were highly beneficial in terms of performance, it is seldom done because of the difficulties it begets. In order to remove this barrier, we propose a set of tools that will ease the development of new, safe and efficient schedulers: a domain-specific language (DSL), a verifier and a new feature in Linux, scheduler hot-plugging.

We first propose **Ipanema**, a **DSL** tailored to write scheduling policies. The idea is to allow developers that are not expert in kernel programming to write custom policies for their applications. By design, the language exposes high-level abstractions to the developer so that he can focus on the scheduling policy rather than on the difficulties of low-level kernel programming.

When developing a scheduler, as well as any other software, ascertaining that the code does what the programmer wanted is useful. This means that we should be able to verify a set of properties on the code, like the ones presented in Section 2.3.5. We propose a **verifier** to formally verify scheduling properties on the policies written in Ipanema. Our DSL approach and our tool chain allow us to ease the verification process of such properties.[33]

[32] For Linux, that would be `gdb` and `kgdb`, as well as various monitoring facilities like `ftrace` and `perf`.

[33] In this thesis, we will not go into details on the verification aspects of this work.

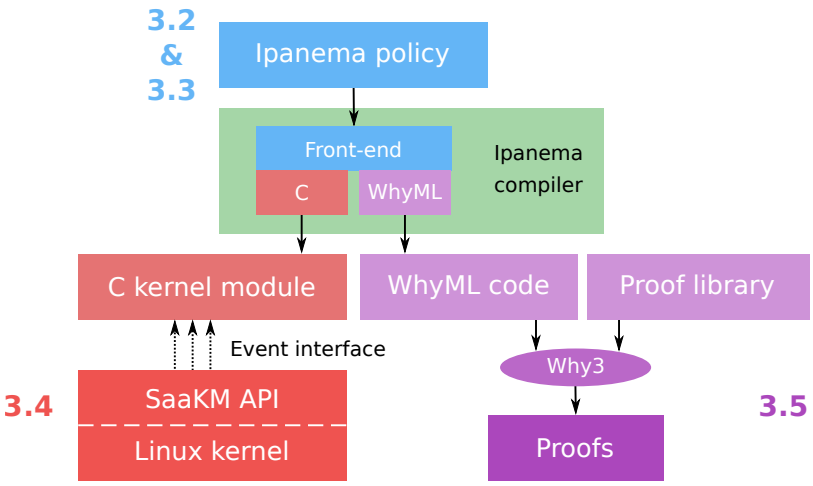


Figure 3.1: The Ipanema tool chain, with the DSL in blue, the execution system in red and the verification system in purple. Related sections of this chapter are also indicated.

Finally, to ease the simultaneous use of multiple custom policies, we implement a new feature in Linux, **Scheduler as a Kernel Module**, or `SaaKM`. This new feature, in the form of a scheduling class, allows users to insert and remove scheduling policies at run time. This helps in the development phase by reducing compile times since it is no longer necessary to recompile a complete kernel binary every time the scheduler is modified. For end users, it allows to easily change the scheduler used for different applications at will.

3.1 THE IPANEMA TOOL CHAIN

As explained before, we have two objectives: easing the development process of schedulers and allowing to prove properties on the scheduling algorithm. To achieve this, we use the tool chain presented in Figure 3.1. The tool chain is composed of three parts glued together by the Ipanema compiler (in green): the DSL (in blue), the execution system (in red) and the verification system (in purple).

THE DSL. Scheduler developers write scheduling policies with the **Ipanema DSL**. The language and the abstractions behind it will be presented in Section 3.2, followed by code excerpts from two policies in Section 3.3. Policies are then passed to the **Ipanema compiler**'s front end that will translate it into an internal representation. At this stage, early safety checks are performed to enforce rules set by the design of the DSL. From this internal representation, the compiler generates code for two different targets: C and WhyML.

EXECUTION SYSTEM. The C target is used to generate a Linux kernel module, in C, for the **execution system**. This module complies

with a specific API we define, `SaaKM`. It must be compiled with a standard C compiler like `gcc` and inserted at run time in a custom Linux kernel. This customized kernel is a standard kernel with an additional scheduling class, `SaaKM`, with an event-based API close to Ipanema's events, that allows policies to be compiled separately from the kernel and added at run time. The `SaaKM` API and policy management tools are presented in Section 3.4

VERIFICATION SYSTEM. The second target generates WhyML code, an ML-like imperative language supported by the state-of-the-art Why3 program verification platform [20]. This WhyML code is then used in the **verification system**, in addition to a hand-written proof library. This library consists of proof skeletons where the generated WhyML code can be inserted at specific locations. The resulting WhyML code is then executed by Why3. If the verified property does not hold, Why3 produces a counter-example that exhibits a violation of the property. We briefly present the verification aspects of this work in Section 3.5 but we do not dwell on this subject since it is out of the scope of this thesis.

RELATED WORK. While this DSL-based approach with an execution and a verification back end is novel in schedulers, similar approaches were proposed for other subsystems. Cogent [4] is a DSL aimed at writing file systems. It uses a similar approach with two back-ends, one for execution and one for verification.

3.2 THE DOMAIN-SPECIFIC LANGUAGE APPROACH

To fulfill both objectives of ease of development and property verification previously stated, we choose the DSL approach. The restrictions enforced by design in the language will prevent developers from introducing bugs in their code. The abstractions exposed by the DSL allow developers to fully focus on the scheduling aspects of their development by not having to worry about low-level issues. Furthermore, the compiler automatically generates the generic parts of the code. For example, concurrency is a major difficulty in programming, all the more so in kernel code. With the DSL approach, we can leverage the design of our language to automatically generate lock management code, and prevent illegal lockless modifications to shared variables.

Another good property of the DSL approach is to ease the generation of proofs based on the code. Indeed, the design of the language forces the developer to write its code in a certain way. The ensuing uniformity in the written code makes it easier to extract parts of the code and insert them in proof skeletons.

Instead of developing a DSL from scratch, we build upon the Bossa [127] DSL that targets the development of schedulers on sin-

	Event	Description
Thread events	`new`	Thread creation with `fork()` or `clone()` system call.
	`tick`	Periodic clock tick when a thread is running.
	`schedule`	Thread election.
	`yield`	Voluntary yielding with the `yield()` system call.
	`block`	Thread is not runnable anymore (I/O, sleep, ...).
	`unblock`	Thread wakes from a blocked state.
	`exit`	Thread termination.
Core events	`balancing`	Periodic rebalancing between cores. Triggered after every tick.
	`newly_idle`	No runnable thread available. Last opportunity to avoid idleness by stealing threads on other cores.
	`enter_idle`	No runnable thread available and thread stealing failed.
	`exit_idle`	New runnable thread available on an idle core.
	`core_entry`	Core insertion (at startup or with vCPUs).
	`core_exit`	Core removal (at shutdown or with vCPUs).

Table 3.1: List of Ipanema events sorted by category (thread or core).

gle core machines. We therefore extend Bossa to handle multi-core machines and develop the **Ipanema** DSL. In this section, we give an overview of the Ipanema DSL, detail the abstractions used in Ipanema and present some policies.

3.2.1 *The Ipanema Language*

The Ipanema DSL, as its predecessor Bossa, is an event-based language. The developer writes a set of handlers that will be called upon when something happens on the system that necessitates the scheduler to act. There are two categories of events: thread and core events. Thread events are related to a specific thread while core events are related to a specific core. Table 3.1 lists these events. For example, when a thread performs a blocking I/O operation, the `block` event will be triggered and the corresponding code in the Ipanema scheduling policy will be executed. Each event has a particular semantic enforced by design. This semantic can be represented as two finite state machines: one for threads and one for cores. These finite state machines will be detailed in Section 3.2.2.

Basically, a developer writing a new scheduling policy with Ipanema will define the thread and core attributes and event handlers. Thread attributes are per-thread variables solely used by the policy. For example, if one implemented CFS in Ipanema, three attributes would be needed: `vruntime`, `weight` and `load`.[34] As for core attributes, we would

[34] See Section 2.5.2.

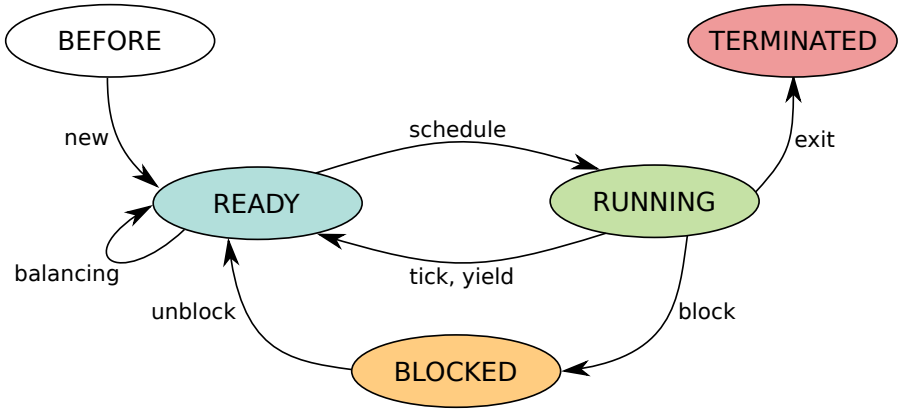


Figure 3.2: Thread finite state machine in Ipanema.

need to at least maintain a runqueue for READY threads and a reference to the RUNNING thread. In addition, one could maintain the core's load as the sum of its threads' loads. Events would then use these attributes to change threads' states, with respect to the abstractions presented in Section 3.2.2.

A considerable advantage of using a DSL compared to C code is that complex operations can be handled automatically. For example, the compiler is able to automatically generate lock management on shared variables. Illegal situations can also be detected at compilation time and therefore avoided. For example, when a thread completes a blocking I/O operation, it should not be placed in a BLOCKED state again, but in a runnable state.

3.2.2 *Abstractions*

As explained previously, Ipanema features a set of abstractions that define the behavior of multiple components of the scheduler. They help constrain what the developer can or cannot do with the language. They also ease the process of proving properties. Ipanema provides four major abstractions: threads, cores, topology and load balancing.

THREADS. **Threads** are defined by a finite state machine greatly inspired by the one presented in Section 2.3.2. Figure 3.2 shows the one used in Ipanema. The states are the same as the generic three-state thread model presented earlier, with the addition of two new states: BEFORE and TERMINATED that represent threads before their creation, i. e. while being initialized, and after their termination, i. e. zombies. All transitions are tagged with one or multiple events. This means that this transition can only be performed during these specified events. Conversely, an event can only happen during a transition that is tagged by it.

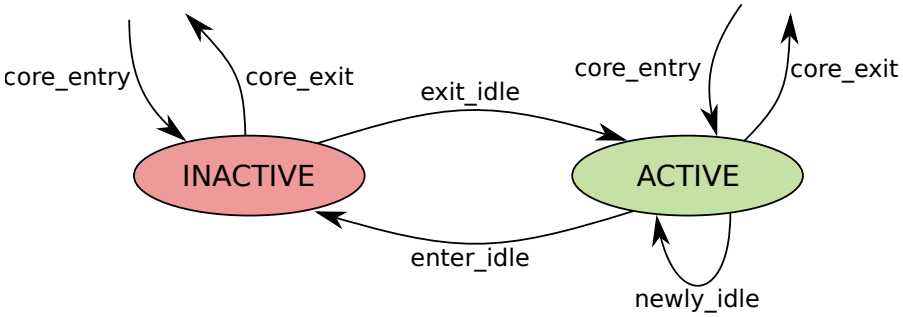


Figure 3.3: Core finite state machine in Ipanema.

CORES. **Cores** are also defined by a finite state machine pictured in Figure 3.3. Currently, we only have two states, ACTIVE and INACTIVE cores, and transitions also correspond to Ipanema events. ACTIVE cores have work to perform, i. e. there is at least one READY or RUNNING thread on this core. On the other hand, INACTIVE cores have no work to perform, and failed to steal work from other cores during the `newly_idle` event. We also allow cores to "appear" and "disappear" of the machine with the `core_entry/exit` handlers. These handlers are used for two reasons:

- when the policy is inserted (resp. removed) in the kernel, each core is initialized (resp. destroyed) with the `core_entry` (resp. `core_exit`) handler,
- when running in a VM, the hypervisor can add or remove vCPUs.

HARDWARE TOPOLOGY. **Topology** is also an important abstraction that allows scheduling policies to manage threads while accounting for SMT, cache locality and NUMA architectures. Our topology abstraction is influenced by the one of Linux. Cores are divided into domains that are organized in a hierarchical way, as shown in Figure 3.4. Each domain is tagged with a set of flags that express the relationship between the cores of a domain: `SMT` means that cores of this domain share computing hardware, `LLC` means that they share a last-level cache and `NUMA` means that accesses to memory are not uniform in this domain. In our DSL, we allow developers to redefine the topology for their needs. For example, the second level of the topology in Figure 3.4, tagged as `LLC`, could be removed if we wish to ignore cache locality altogether.

LOAD BALANCING. **Load balancing** is a particular event in that it involves multiple cores at the same time. This can lead to complex locking mechanisms that must be hidden from the developer. To do so, we split load balancing into three phases as depicted in Figure 3.5.

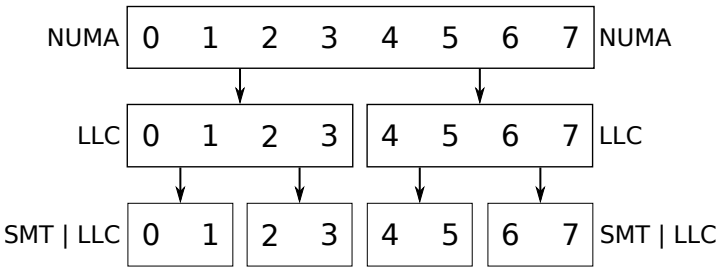


Figure 3.4: Topology of an 8-core 2-node NUMA machine in Ipanema.

PHASE 1. When a balancing event is triggered on core `dst`, it will first look at all cores and check which ones have stealable work. In this first phase, the developer only writes the `can_steal_core()` function. For example, we could define a core as stealable if it has more threads than `src`, using core attributes defined in the Ipanema policy. Stealable cores are stored in the `stealable_cores` set.

PHASE 2. A core must then be selected among these stealable cores. The developer only writes the `select_core()` function that does the choice and the `stop_steal` condition to abort balancing.

PHASE 3. Finally, the actual stealing takes place. First, threads are removed from the targeted core with the core's lock held. Stealable threads are chosen with the `steal_thread()` function supplied by the developer, and stealing continues until the condition `stop_steal_core` becomes true. When enough threads have been removed, the lock on the target core is released. Then, a lock is taken on `src` and all stolen threads are added to its runqueue. Note that there might be incoherence between the observation phases 1 and 2, and the action phase 3. Indeed, the first phases are performed lockless, which means that the data used to choose a core to steal from can change before the stealing is actually done. Phase 3 might therefore fail if, for example, no threads are available anymore due to blocking events happening concurrently.

The three phases described here suppose that we check all cores of the machine and do not account for hardware topology. Ipanema does support hierarchical balancing, following the topology defined by the developer, by adding two additional phases analogous to phases 1 and 2 at the domain level: `can_steal_group()` and `select_group()`. They are also performed with no locks held. The former filters the domains from which threads could be stolen, mirroring phase 1, while the latter chooses the domain from which threads will be stolen, mirroring phase 2. We won't detail these phases further because they behave in the exact same way as phases 1 and 2, except they operate on domains instead of cores.

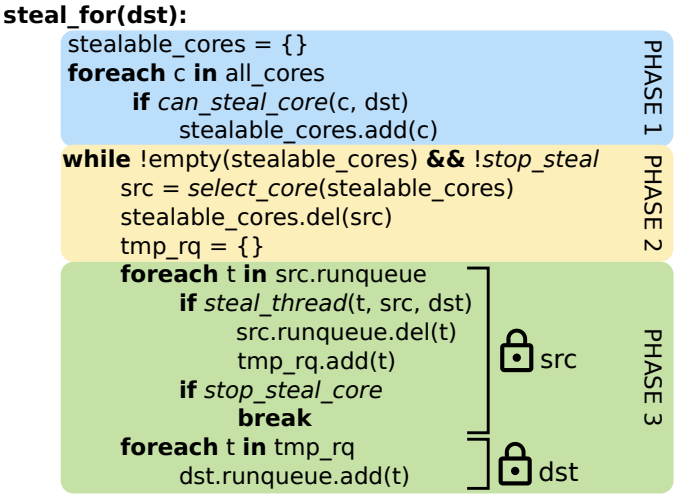


Figure 3.5: Load balancing phases in Ipanema.

3.3 THE IPANEMA DSL THROUGH POLICIES

Now that we have defined the abstractions upon which Ipanema is built, let's see practical examples of scheduling policies written with our DSL. In this section, we will present various aspects of the Ipanema language through policies we implemented and tested.

3.3.1 *CFS-like Policy*

First, we present excerpts of a CFS-like policy, `cfs.ipa`. This policy is a simplified version of the default Linux scheduler. We first present the thread and core attributes specific to this policy. We then present some event handlers of this policy.

ATTRIBUTES. Listing 3.1 presents the definition of a thread and a core in this policy. Threads are defined (lines 2–7) by their *load*, *vruntime*, last date of scheduling (*last_sched*) and the core they are on (*cpu*). The `system` keyword means that this field is not managed by the developer but automatically modified in the generated C code. In this case, the *cpu* field always references the core where the thread is currently located, with no need to explicitly modify it.

Cores are defined in two parts: the definition of per-core metadata (lines 10–13) and how threads are stored (lines 15–22). In terms of metadata, a core is defined by its *id* (attributed by the system), its load (*cload*), the smallest *vruntime* that a thread has on this core and the set of domains the core is a part of. Threads are stored into four categories, each one tagged by a state defined in the thread finite state machine. There is a single RUNNING thread, `current`, and a set of READY threads sorted by *vruntime* in ascending order. This set is implemented as a red-black tree for performance, but Ipanema allows for other data

structures such as FIFO queues. The last two categories, BLOCKED and TERMINATED, are untyped because no thread is stored in them.

```
1  // Attributes
2  thread = {
3     time vruntime;
4     time last_sched;
5     int load;
6     system core cpu;
7  }
8
9  core = {
10    system int id;
11    int cload;
12    set<domain> sd;
13    time min_vruntime;
14
15    threads = {
16       RUNNING thread current;
17       shared READY set<thread> ready:order = {
18          lowest vruntime
19       };
20       BLOCKED blocked;
21       TERMINATED terminated;
22    }
23 }
```

Listing 3.1: Thread and core definitions in the `cfs.ipa` policy.

EVENTS. With these attributes defined, we can now write the event handlers. Listing 3.2 presents some handlers for this policy. The variable `target` represents the thread concerned by the event, `now()` and `first()` are helper functions provided in the standard library of the language. They respectively return the current time in nanoseconds and the first element of a set, with respect to the order specified, i. e. in this case, the thread with the lowest *vruntime* in the READY state.

The `tick` event (lines 2–10) checks if the thread has used all its time slice. If not, the thread continues its execution; else, the metadata of the thread is updated with user-defined functions not represented here, and the thread is placed in the READY runqueue, and removed from the RUNNING state. Upon detecting the absence of a RUNNING thread, the `schedule` event will be triggered.

The `block` event (lines 12–17) updates the blocking thread's metadata in the same fashion as the `tick` event and places the thread in the BLOCKED state. In this policy, this state is untyped, and therefore not backed by a data structure. This means that a BLOCKED thread cannot be referenced until it wakes up, in the `unblock` event.

The `schedule` event (lines 19–25) retrieves the first thread in the READY state, updates some metadata and makes it the currently running thread. This thread will now be able to use the core and execute its code upon returning to user space.

```
1  // Event handlers
2  On tick {
3      time curr_quanta = now() - target.last_sched;
4
5      if (curr_quanta > max_quanta) {
6          update_thread(target);
7          update_load(target);
8          target => ready;
9      }
10 }
11
12 On block {
13     update_thread(target);
14     update_load(target);
15
16     target => blocked;
17 }
18
19 On schedule {
20     thread p = first(ready);
21
22     p.last_sched = now();
23     min_vruntime = p.vruntime;
24     p => current;
25 }
```

Listing 3.2: A subset of the events of the `cfs.ipa` policy.

3.3.2 *ULE-like Policy*

Listing 3.3 shows an excerpt of a ULE-like policy in Ipanema. Through this policy, we present the load balancing abstraction and how it translates into Ipanema code.

First, we need to present some thread and core attributes used in load balancing. Lines 1–25 show the definitions of threads and cores. The important thing to note regarding threads is that their *load* always equals 1. We use the *load* to count the number of threads on the core because ULE uses this metric to balance cores. Other attributes, like *slptime*, are mainly used to determine priority.

Cores are defined by an *id* and a *cload*, as in the CFS-like policy. In addition, a boolean, *balanced*, is used during load balancing to determine if a core has recently participated in a work stealing operation, be it as a source or a target. Threads are stored similarly to the CFS-like policy, except for READY threads. They are stored in two queues, *realtime* and *timeshare*, implemented as FIFO doubly linked lists. Threads with the `REGULAR` priority (see line 4) are stored in the latter, while all other threads are stored in the former. When a `schedule` event is triggered, the *realtime* queue is always checked before the *timeshare* queue.

```
1  thread = {
2     system thread parent;
3     int load = 1;
4     int prio;       /* INTERRUPT, REGULAR, INTERACTIVE */
5     core last_cpu; /* last cpu the thread executed on */
6     int slice;      /* timeslice */
7     time rtime = ticks_to_time(0);    /* runtime */
8     time slptime = ticks_to_time(0);    /* sleep time */
9     time last_blocked = ticks_to_time(0); /* last block */
10    time last_schedule = ticks_to_time(0);/* last schedule */
11 }
12
13 core = {
14    system int id;
15    int cload;
16    bool balanced;
17
18    threads = {
19       RUNNING thread current;
20       shared READY queue<thread> realtime;
21       shared READY queue<thread> timeshare;
22       BLOCKED set<thread> blocked;
23       TERMINATED  terminated;
24    }
25 }
26
27 steal = {
28    can_steal_core(core src, core dst) {
29       dst.balanced ? false :
30         src.balanced ? false :
31           src.cload > dst.cload
32    } => stealable_cores
33
34    do {
35       select_core() {
36          first(stealable_cores order = { highest cload })
37       } => busiest
38
39       steal_thread(core here, thread t) {
40          if (busiest.cload - here.cload >= 2) {
41             here.balanced = true;
42             busiest.balanced = true;
43             if (t.prio == INTERRUPT || t.prio == INTERACTIVE)
44                t => here.realtime;
45             else
46                t => here.timeshare;
47          }
48       } until (here.balanced)
49    } until (true)
50 }
```

Listing 3.3: Load balancing in the `ule.ipa` policy.

The load balancing operations are defined in lines 27–50. We can see the three phases defined in Section 3.2.2 with the `can_steal_core()`, `select_core()` and `steal_thread()` functions. The first phase excludes cores that already participated in a load balancing and cores

that have fewer threads than the stealing core. The second phase selects the core with the largest number of threads among the remaining cores. Finally, the last phase steals a single thread if and only if the imbalance between both cores decreases after the thread migration, with a priority given to *realtime* threads. Note that phases 2 and 3 are surrounded by a loop because some policies could need to steal threads from multiple cores. Here, the condition always breaks the loop, so only a single core can be targeted.

3.4 SCHEDULER AS A KERNEL MODULE

As presented in Section 2.4.4, the Linux kernel provides a way to develop schedulers: the scheduling class internal API. In this section, we present the scheduling class we implement in Linux to allow users to hot plug schedulers at run time as kernel modules instead of having them built-in the kernel binary. We also present how to interact with this new scheduling class and use it.

3.4.1 *Extending the Scheduling Class Model for the Linux Kernel*

First, we assess the limitations of the current internal API of Linux, the scheduling class API. Then, with these limitations in mind, we propose a new internal API, Scheduler as a Kernel Module or `SaaKM`, that will allow us to easily use new scheduling policies in Linux.

3.4.1.1 *Limitations of the Current Internal API*

The scheduling class internal API allows developers to implement schedulers in Linux, as presented in Section 2.4.4. However, schedulers developed with this API must be embedded in the kernel binary. Schedulers compiled in the binary are always present and cannot be added or removed at run time from the system. This limitation precludes developers to distribute new scheduling policies easily, they must distribute a complete kernel binary. This also means that, for end users, adding schedulers to Linux increases the size of the kernel binary. For developers, the development process becomes more tedious because a complete kernel must be recompiled for each modification. When debugging, this quickly becomes inconvenient.

Second, when studying the scheduling class internal API, we felt like the API was not developed to enable the implementation of any scheduler, but only to accommodate for the existing schedulers in Linux. Indeed, handlers are not precisely specified, with a sparse documentation that does not cover them all.[35] For example, the `enqueue_task()` handler is documented as follows:

[35] As of Linux v5.4, only 7 out of 24 handlers are documented, and quite briefly.

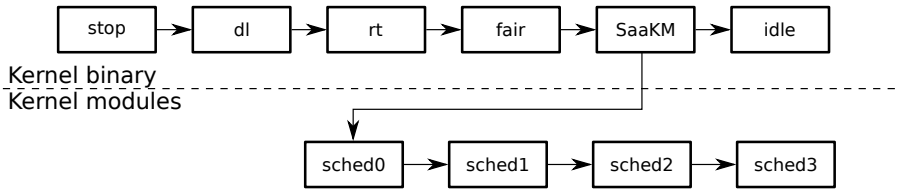


Figure 3.6: Architecture of the scheduler subsystem in Linux with our new `SaaKM` scheduling class.

> *Called when a task enters a runnable state. It puts the scheduling entity (task) into the red-black tree and increments the nr_running variable.*

However, this description is biased by the implementation of CFS, as other scheduling classes do not necessarily use red-black trees. Moreover, this description seems to indicate that the handler should insert a single thread into the runqueue. However, this is not really the case since the `SCHED_DEADLINE` policy does not enqueue the thread under specific conditions.[36]

[36] When the thread's allocated time slice is exhausted.

3.4.1.2 *The `SaaKM` Scheduling Class*

We choose to implement a new scheduling class that allows users to implement thread schedulers as kernel modules and plug them into the scheduler subsystem. Figure 3.6 shows the new architecture of the scheduler subsystem with our new scheduling class. As previously explained in Section 2.4.4, scheduling classes are organized in a singly linked list, sorted by priority. Now, when the scheduling subsystem needs a new thread to schedule, i. e. the `pick_next_task()` function is called, it will, as usual, iterate through each scheduling class to check if a runnable thread is available. When the `SaaKM` scheduling class is reached, it will iterate through each of its registered scheduling policies (kernel modules) to find a runnable thread. If none is found, the `idle` thread will be scheduled. We use a static priority between our scheduler modules, as is done originally in Linux. Implementing a form of *meta-scheduling* is a possible future work we did not explore.

`SaaKM`'s PRIORITY

We place `SaaKM` policies at the lowest priority, excluding the `idle` class. The reason behind this is that we do not want to hinder the normal behavior of the system when not using custom schedulers. Additionally, multiple subsystems in Linux use CFS and assume that only real-time threads have a higher priority. Using a `SaaKM` policy that would hog some CPUs might cause some essential kernel threads to be starved from CPU time and break the system. One example of such a thread is the one managing the read-copy-update (RCU) synchronization mechanism in the kernel.

	Function	Description
Thread events	`new_prepare(thread)`	Called when a thread is created, with the `fork()` or `clone()` system calls. Returns the core where `thread` should be placed.
	`new_place(thread, core)`	Called after `new_prepare()`, with `core` locked. Must place `thread` on a runqueue of `core`.
	`tick(thread)`	Called when a clock tick happens with `thread` running.
	`yield(thread)`	Called when `thread` uses the system call `yield()`.
	`block(thread)`	Called when `thread` must block (I/O, sleep, ...). Must remove `thread` from its runqueue.
	`unblock_prepare(thread)`	Called when `thread` wakes from a blocked state. Returns the core where `thread` should be placed.
	`unblock_place(thread, core)`	Called after `unblock_prepare()`, with `core` locked. Must place `thread` on a runqueue of `core`.
	`terminate(thread)`	Called when `thread` terminates. Must remove it from any runqueue.
Core events	`schedule(core)`	Called when a new thread should be scheduled on `core`, i. e. there is no running thread.
	`balancing(core)`	Called periodically (at every tick) for periodic rebalancing.
	`newly_idle(core)`	Called after `schedule()` if no thread is available for scheduling. May try to steal threads from other cores.
	`enter_idle(core)`	Called after `newly_idle()` if `core` still has no thread to run. Must put `core` in idle state.
	`exit_idle(core)`	Called when `core` is idle and a new thread becomes available on `core`. Must put `core` in active state.
	`core_entry(core)`	Called when a new core appears on the machine. Should setup `core`.
	`core_exit(core)`	Called when `core` disappears from the machine. Should move out all threads from `core` and put them on other cores.

Table 3.2: List of functions in the `SaaKM` internal API.

In order to be available to threads, a scheduler module must register itself with the SaaKM scheduling class. To do so, it must implement the set of functions defined in Table 3.2. The SaaKM scheduling class will then call each of these functions at the right time, holding the right locks, to allow the scheduler module to perform the needed operations. These events are split into two categories, *thread events* and *core events*. This is a similar split as the one we defined for the events of the Ipanema DSL in Section 3.2.

For example, when a thread is created, be it with the `fork()` or the `clone()` system call, the `new_prepare()` function will be called with the lock on the `struct task_struct` held. This handler is the first function of the scheduler module called when a thread starts using this scheduler. This means that a scheduler module should initialize the per-thread metadata it will use for scheduling. This handler must also return the core on which the newly created thread should be placed. Indeed, a newly created thread has not been allocated a core to be scheduled on yet. When this core is returned, the scheduling class and the common part of the scheduler subsystem will perform generic operations and lock the chosen core. Then, the `new_place()` handler will be called, and the scheduler module will be responsible for placing the newly created thread on the core returned by the `new_prepare()` handler, in a runqueue. The scheduler subsystem will then release both the core's and the thread's respective locks.

3.4.2 *Live Management of Schedulers*

With a SaaKM-enabled kernel, users can use schedulers in the form of kernel modules that can be inserted and removed at run time. Threads can then use these newly inserted schedulers through different means:

- a **system call** already available in Linux, `sched_setattr`, in order to do it programmatically inside the code,
- **user space tools** we provide, `saakm_start` and `saakm_setpolicy`,
- a **cgroup** interface specific to SaaKM.

But first, before using a policy, users must compile the kernel module through the Linux kernel's build system, using the `modules` rule of the kernel's Makefile.

KERNEL INSERTION AND REMOVAL. As soon as your kernel module containing your scheduling policy has been compiled, you can insert it at run time with the `insmod` shell command. After doing this, you can check the list of available SaaKM policies by reading the `/proc/saakm/policies` file. Let's assume that we have three policies named *policyA*, *policyB* and *policyC*. Inserting them and reading the content of the aforementioned file will give this output:

```
redha@localhost:~:$ insmod policyA.ko policyB.ko policyC.ko
redha@localhost:~:$ cat /proc/saakm/policies
0 policyA 0
1 policyB 0
2 policyC 0
```

Each line contains an identifier, the name of the policy and the number of threads currently using the policy. The identifier is used when switching to a given policy. As for the removal of a policy, it is performed with the `rmmod` shell command. Note that removing a policy will fail if any thread is currently using the policy.

THE SYSTEM CALL INTERFACE. Linux contains multiple scheduling classes to choose from, and provides a set of system calls that allow threads to switch to and from each of them. These system calls ultimately use the same code, but offer a different prototype. The most complete one is `sched_setattr`. We can use it to switch to the `SaaKM` scheduling class and provide the identifier of the policy we wish to use. This is how it can be used to switch a thread to *policyB*:

```
struct sched_attr attr = {
        .sched_policy = SCHED_SAAKM,
        .sched_saakm_policy = 1      // id of policyB
};

sched_setattr(0, &attr, 0);
```

THE USER SPACE TOOLS. If a complete program should be launched with a `SaaKM` policy, we also developed a user space tool, `saakm_start`. This tool uses the `sched_setattr` system call to switch to the policy passed as an argument before starting the program. In order to launch the `ls` program with *policyB*, one would run:

```
redha@localhost:~:$ saakm_start 1 ls
foo/ bar/ file.txt
redha@localhost:~:$
```

If a user wants to change the policy of a currently running thread, the `sched_setattr` system call permits to do it. The first parameter of this system call is the process identifier (PID) of the thread that should change its scheduling policy. In `saakm_start`, we used the value 0 in order to change the current thread's scheduling policy. The `saakm_setpolicy` user space tool we provide is a wrapper for this system call. It takes two arguments: the target PID and the identifier of the destination `SaaKM` policy. Let's say we want the thread with the PID 1337 to use *policyC* as a scheduler. We would do the following:

```
redha@localhost:~:$ saakm_setpolicy 1337 2
redha@localhost:~:$
```

THE CGROUP INTERFACE. Managing a large number of threads with the previous tools is cumbersome. Moving a bulk of threads from one policy to another would mean to move each thread individually from one policy to another. To facilitate this process, needed for the performance-driven feature search approach presented earlier in Section 5.4.2, we implement a *cgroup* interface for `SaaKM`.

The usage of this interface is simple: users need to create a directory in the cgroup pseudo file system, usually located in `/sys/fs/cgroup/`, under the `SaaKM` directory. The newly created directory will already contain the `saakm.policy_id` and `tasks` files. The former is used to write the `SaaKM` policy identifier to be used in this *cgroup*.[37] The latter is used to read or write the list of threads in this *cgroup*, i. e. their PIDs. Writing a list of PIDs in this file will trigger a change of scheduling policy in bulk, with a single `write` system call.

[37] A value of -1 is equivalent to the use of CFS.

3.5 PROPERTY VERIFICATION

In addition to the Ipanema DSL and the execution system, `SaaKM`, we also collaborate with researchers from the University of Sydney and CNAM[38] to provide a facility to formally verify properties on the scheduling algorithm. As previously shown in Figure 3.1, this verification system consists of a compiler back end and a proof library that, when used together with the Why3 platform [20], verify scheduling properties. In this section, we will provide an overview of the verification of the work conservation property. We will not provide a detailed explanation of the formal verification part since it is out of the scope of this thesis. We first explain the general verification process and then give some insights about how we verified work conservation.

[38] Conservatoire National des Arts et Métiers, Paris, France

3.5.1 *Overall Process*

The general idea of the verification process is to write a skeleton of the proof in WhyML, the language processed by Why3. This skeleton contains all the generic parts of the proof. The specific parts of the proof should be extracted from the Ipanema code and inserted in the skeleton. This is done by the second back end of the Ipanema compiler that generates WhyML code. The completed WhyML code is then passed to the Why3 program that will check if the property holds. If it does not, it will provide a set of states that violate the property, i. e. a counter-example.

3.5.2 *Work Conservation*

As defined previously in Section 2.3.5, work conservation is the property that if a core on the machine is overloaded, i. e. has more than one thread, then no core is idle, i. e. has nothing to run. The property must hold at all times, i. e. after any scheduling event. For events that only affect a single thread, e. g. thread creation or unblock, we only want to check that if the destination core is overloaded, then no core is idle. For events that can affect multiple threads, e. g. load balancing, we want to be sure that if **any** core on the system is overloaded, no core is idle.

Both definitions describe **instantaneous work conservation** (*WC*). However, on a real multi-core system, events are happening concurrently. It is therefore difficult to ensure that no core is idle if cores can become idle simultaneously. This is especially true for load balancing where long critical sections in mutual exclusion would be needed. This is not possible if we want to keep a good level of performance.

A possible workaround would be to prove **eventual work conservation** (*EWC*). This property specifies that if events no longer occur, i. e. the system is stable, work conservation will eventually be achieved in the future. On real systems, such a stable situation never happens. This solution also fails to capture situations where work conservation violations are due to events happening concurrently.

To work around the problems of both properties, we propose a stronger property than eventual work conservation, **concurrent work conservation** (*CWC*). This property uses slightly different definitions of overloaded and idle that account for concurrent blocking and unblocking events. An overloaded core during a scheduling event is a core with more than one thread where no unblocking event happened during the scheduling event studied. A similar redefinition of an idle core is also provided.

Instantaneous work conservation is the strongest property of all three. Proving it proves the other two. Conversely, eventual work conservation is the weakest, meaning that proving it does not prove the other two. Therefore:

$$WC > \mathbf{CWC} > EWC$$

We choose to prove *CWC* because it is the strongest property we can prove that has a practical use.

We use the constrained structure of the Ipanema DSL to ease the proving effort. The events and the phases of load balancing give a prior knowledge of the locking scheme of the scheduler. With this information, we know when the state of the system is consistent and when it is not. For a more detailed description of the proof, please refer to this published work [105].

3.5.3 *Implications of Verification on the DSL*

As previously hinted, the Ipanema DSL was designed with verification in mind. The constraints we force on developers have the objective of easing the verification process. For example, the load balancing is split into three phases in order to inject smaller more specific pieces of code into the proof skeletons. Even if this leads to a bit more complexity for the developer, the gains outweigh the losses. In our opinion, the ability to be sure that a scheduling property holds is worth a few more lines of code. For example, a work conservation bug in ULE, the scheduler of FreeBSD, remained undetected for three years before being fixed [22]. In Linux, work conservation bugs were also found by Lozi *et al.* [113].

3.5.4 *Related Work*

The conventional approach to improve the correctness of kernel code is **testing**. The Linux Testing Project [108] and community testing are used to detect bugs in the Linux kernel. Additionally, the Linux Kernel Performance project [33] is used to detect performance regression problems, and other tools detect abnormally long system calls [138, 157]. With these tools, a large number of bugs or regressions have been detected. However, they cannot be used to detect subtle bugs that rarely happen, such as concurrency bugs or hardware-specific bugs.

Model checking has also been used to find bugs that lead to crashes [128] or deadlocks [189]. Using model checkers on schedulers is challenging because of the combinatorial blow up of the state space due to the number of scheduling events that happen on a machine with a large number of cores and threads.

Finally, in the recent years, **formal verification** of OSs is becoming a large trend. Formal techniques have been applied to verify complete OSs such as SeL4 [94], CertiKOS [72] or Hyperkernel [129]. It has also been used on more specific parts of OSs such as file systems [31, 32, 159].

3.5.5 *Conclusion*

In order to demonstrate the ability of our DSL to ease the formal verification of scheduling properties, we propose to prove work conservation. In this collaborative work, we provided the definition of the concurrent work conservation property as well as the design of the Ipanema DSL. From now on, if we say that a scheduling policy is proven work-conserving, it means that we proved concurrent work conservation for this policy.

3.6 EVALUATION

In this section, we evaluate policies implemented in Ipanema on multiple applications to determine if Ipanema is suitable to be used in production environments. We compare multiple Ipanema policies inspired by CFS and FreeBSD's ULE to the default Linux scheduler. We also evaluate policies **proven work-conserving** by our verification tool chain. As a reminder, a work-conserving scheduler does not let cores be idle if any core has more than one runnable thread.

In addition to showing that Ipanema policies can be as efficient as policies written directly in C, we also want to demonstrate that we do not add unwanted overhead due to excessive locking for example. More specifically, in the policies proven to be work-conserving, we forbid concurrent load balancing as is performed in CFS. Our evaluation shows that this is not a performance problem on our test applications.

3.6.1 *Experimental Setup*

SYSTEM SETUP. The evaluation is performed on a 4-socket 160-core machine equipped with an Intel® Xeon E7–8870 v4 processor, 512 GiB of memory, running a Debian Buster OS. We remove the effects of dynamic frequency scaling by using the `performance` scaling governor. This governor forces the use of the maximal frequency on all cores.[39] We modify a Linux v4.19 kernel to add support for our Ipanema schedulers through the `SaaKM` interface presented in Section 3.4.

[39] More details on dynamic frequency scaling are available in Chapter 4.

POLICIES. We evaluate our Ipanema tool chain by evaluating five different scheduling policies:

- `CFS` is the vanilla CFS scheduler of Linux v4.19, used as a baseline comparison. It is the only tested scheduler directly written in C.
- `CFS-CWC` is a simplified and slightly modified version of the algorithm of CFS proven to be work-conserving.
- `CFS-CWC-FLAT` is the same algorithm except that it does not account for the hardware topology.
- `ULE` and `ULE-CWC` are simplified versions of the scheduler of FreeBSD, ULE. The latter is slightly modified and proven to be work-conserving.

BENCHMARKS. For our experiments, we use various workloads from the NAS benchmark suite [13], as well as a kernel compilation and a `sysbench` OLTP benchmark [98]. We run all experiments 12 times and present the mean of these runs and their standard deviation.

We exclude I/O intensive benchmarks from the NAS suite because they exhibit a high standard deviation on our machine. We keep only

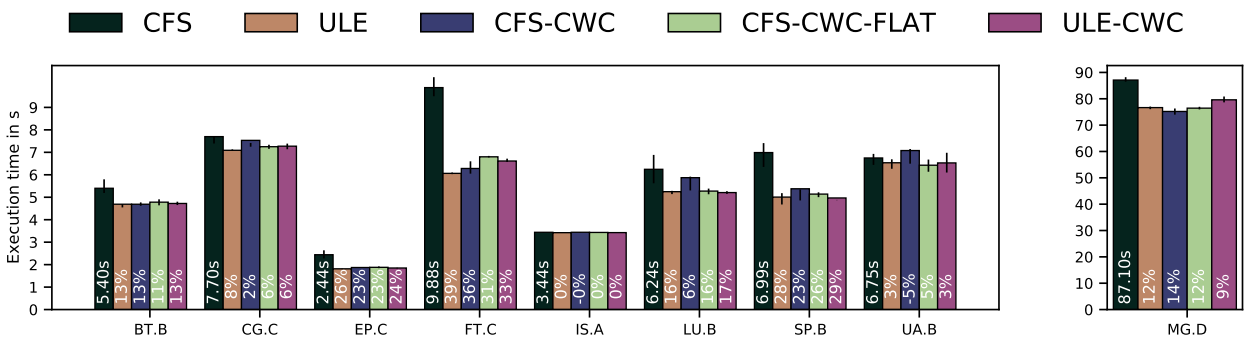


Figure 3.7: Execution time of the NAS benchmarks with 160 threads on CFS and custom Ipanema policies. Lower is better.

applications dominated by computations and synchronization, e. g. barriers. These applications are challenging for our work-conserving schedulers because they contain barriers that wake up a large number of threads simultaneously. In these situations, CFS exhibits work conservation issues that our policies solve.

The two other benchmarks do not exhibit work conservation problems, but they are useful to evaluate potential overheads of Ipanema. They produce a very large number of scheduling events, with threads blocking and waking up constantly. This behavior stresses the scheduler code and locking facilities we implement. A limited overhead on these benchmarks is a good indication of the efficiency of our approach.

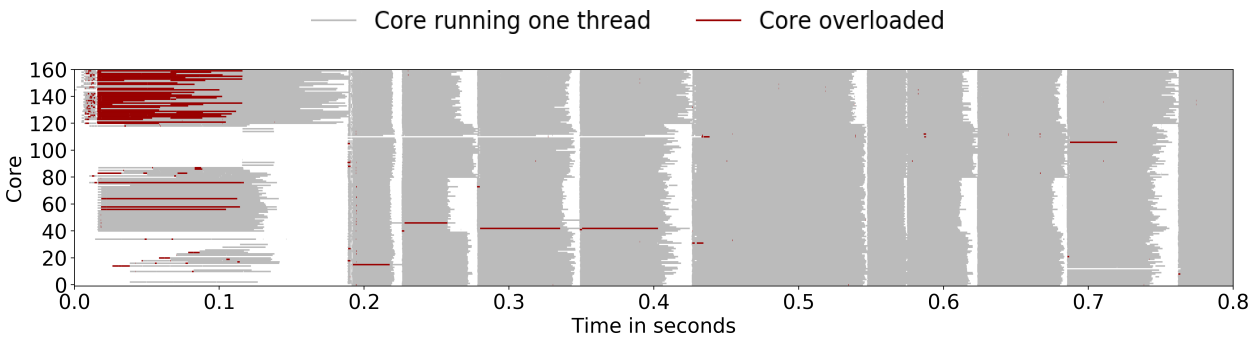
The `sysbench` OLTP is a scriptable database benchmark. We run it with two databases, MySQL 8.0.15 and MongoDB 4.1.8, with a mix of read and write OLTP queries to evaluate request latency and throughput. The benchmark and the database share the machine, and the database is stored in memory in a `ramfs` partition.

The kernel compilation benchmark is a parallel batch workload that compiles the Linux kernel using the `make` program. We compile a minimal kernel, with a configuration generated with the `make defconfig` command. We perform the compilation in memory to avoid long I/Os on the disk.

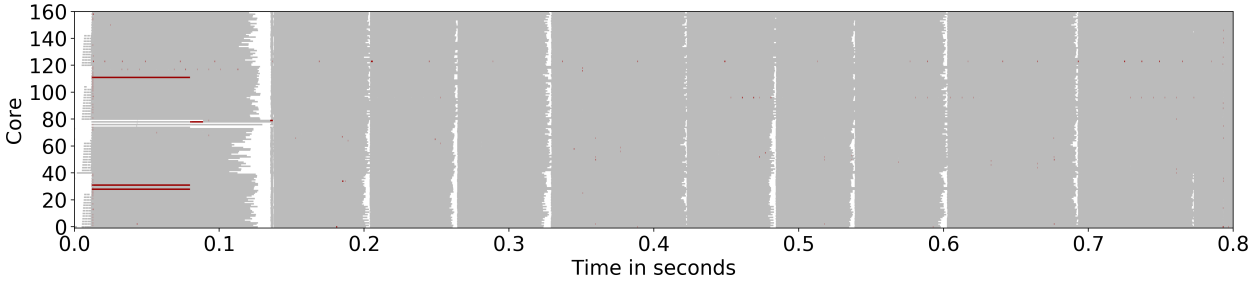
3.6.2 *Performance Results of NAS*

We first evaluate the performance of our Ipanema policies on the NAS benchmarks. Figure 3.7 shows the mean execution time of each benchmark for `CFS` and the performance difference as compared to `CFS` for the Ipanema policies. The `MG.D` benchmark is represented alone for readability because it has a longer execution time than other benchmarks.

Overall, our Ipanema schedulers perform better than vanilla CFS. We compute the geometric mean of all benchmarks for each policy



(a) Execution with vanilla CFS.



(b) Execution with CFS-CWC.

Figure 3.8: First 0.8s of the execution of NAS FT.C

to evaluate the overall improvement. As compared with `CFS`, we observe a 15.5% improvement for `ULE`, 11.8% for `CFS-CWC`, 14.0% for `CFS-CWC-FLAT` and 14.4% for `ULE-CWC`.

The performance improvement of both CFS inspired policies is largely due to work conservation at thread creation time. We verify this by logging the number of threads per core at all times during the benchmark with `ftrace`, a low overhead tracing facility built in the Linux kernel [152]. Figure 3.8 shows a visualization of the logged values for `CFS` and `CFS-CWC` during the first 0.1 s of the execution of the `FT.C` benchmark. Gray lines represent cores with only one thread, red lines represent cores with more than one thread, i. e. overloaded cores. If no line is displayed, the core is idle.

The execution with vanilla `CFS`, represented on Figure 3.8a, shows that cores are overloaded (in red) while other cores are idle (no line) at the beginning of the execution. It takes `CFS` 0.44 s to achieve work conservation, and small placement errors keep violating the property from time to time afterwards. This is due to the NUMA heuristics of CFS that avoid using remote nodes. At the very beginning, all threads are created on a single node (cores 120–159) that is completely overloaded. Shortly after, a second node (cores 40–79) steals a group of threads. It is only after 0.2 s of execution that the remaining two nodes decide to initiate load balancing and steal work.

With `CFS-CWC`, represented on Figure 3.8b, no core is overloaded after the 0.08 s mark. The modifications of this scheduler that make it work-conserving overrule the NUMA heuristics to achieve work

conservation faster. Note that on both figures, we clearly see the barriers, where all cores become idle, and that they are way shorter on our work-conserving scheduler than on CFS.

3.6.3 *Performance Results on Other Workloads*

We now evaluate our Ipanema policies on scheduler-intensive workloads, the compilation of the Linux kernel and `sysbench` OLTP. Figure 3.9 shows all the results.

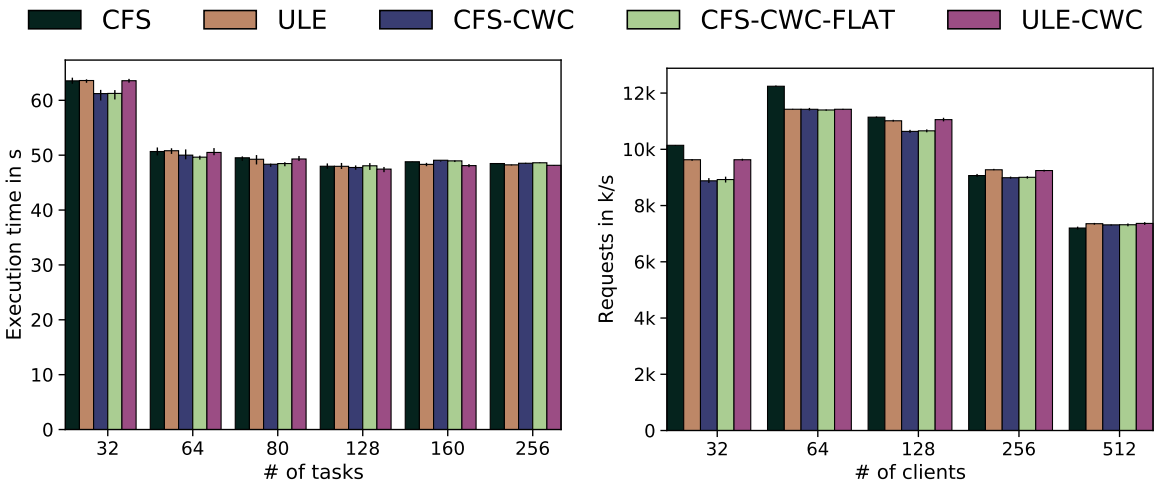
KERNEL COMPILATION. We perform the kernel compilation with different numbers of concurrent jobs, ranging from 32 to 256. Results are shown on Figure 3.9a. On our machine, for all configurations up to 128 jobs, our CFS-like schedulers slightly improve performance as compared to CFS. With more jobs, they become equivalent to CFS. The small gains in performance are explained by the thread placement strategy when a thread is unblocked. The work-conserving policies are more aggressive than CFS when it comes to using idle cores. This is useful on a lightly loaded machine, i. e. with fewer than 160 jobs on our machine. When the machine becomes really loaded, there are fewer idle cores available, so being aggressive has no real impact on performance. Overall, there are no large performance differences between vanilla CFS and our Ipanema policies, with at most a 6% gain with our policies.

SYSBENCH OLTP. We then benchmark a database workload with `sysbench` OLTP on two different database engines: MySQL and MongoDB. These workloads are highly demanding in terms of locking and produce a very large number of scheduling events due to threads frequently blocking and unblocking. Figures 3.9b and 3.9d present the performance in terms of throughput while Figures 3.9c and 3.9e present the 95^{th} percentile of response times. For MySQL, CFS and our Ipanema policies perform quite similarly in terms of throughput and latency, with a maximal difference of 8.2%. For MongoDB, all schedulers also perform similarly, with at most a 3% difference. As with the kernel compilation, these differences are likely due to placement decisions when threads are unblocked.

3.6.4 *A Note on the Code Size*

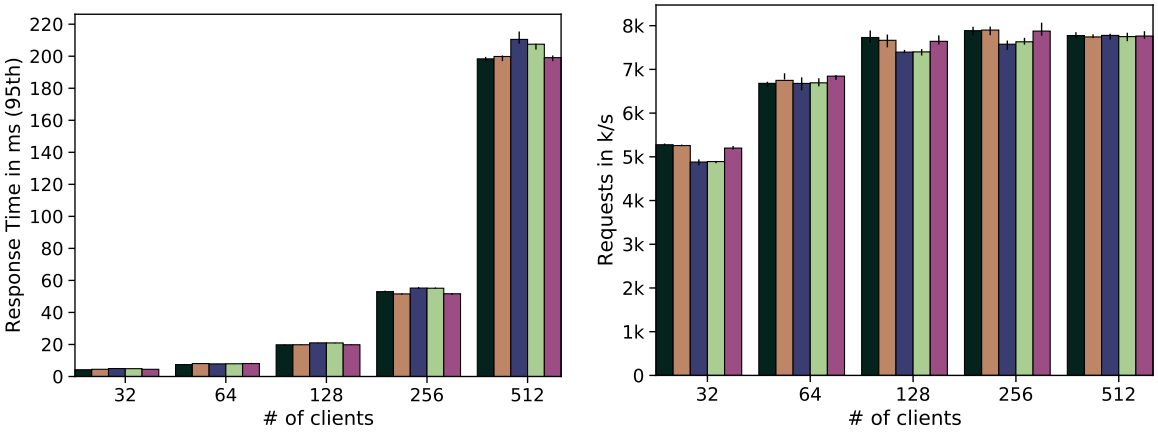
One of the major advantages of using our DSL is the small number of lines of code needed to write a scheduler. Table 3.3 shows the number of significant lines of code of the policies we evaluate.[40] For `CFS`, we show the number of lines of the scheduling policy of the original CFS in the kernel source code. For the Ipanema policies, we show the number of lines in the Ipanema code and the number of lines of the

[40] Blank lines and comments are not included.



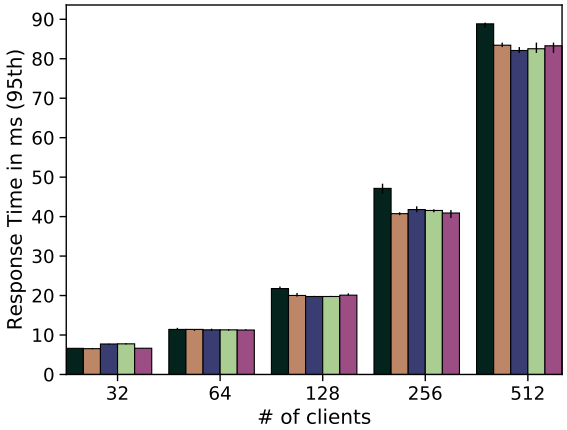
(a) Linux kernel compilation time (lower is better)

(b) sysbench OLTP with MySQL 8.0.15 (higher is better)



(c) sysbench OLTP with MySQL 8.0.15 (lower is better)

(d) sysbench OLTP with MongoDB 4.1.8 (higher is better)



(e) sysbench OLTP with MongoDB 4.1.8 (lower is better)

Figure 3.9: Performance of vanilla CFS, ULE, CFS-CWC, CFS-CWC-FLAT and ULE-CWC.

Policy	Ipanema	C
CFS	✗	5,712
CFS-CWC	360	1,006
CFS-CWC-FLAT	242	791
ULE	272	851
ULE-CWC	245	898

Table 3.3: Size of the code of the tested Ipanema policies in Ipanema and in generated C. The first line, `CFS`, shows the number of lines of the vanilla scheduler of Linux.

generated C code. Our policies are much smaller than the original CFS. This is also the case when comparing to the original code of ULE from FreeBSD that amounts to 2,087 lines. Even after adding the 1,527 lines of code of the `SaaKM` scheduling class, our policies are drastically smaller than `CFS`. In terms of performance, however, our policies are on par or better than `CFS` on most of the applications we tested.

3.7 CONCLUSION

Writing a thread scheduler is a difficult task. It requires knowledge of scheduling algorithms and hardware capabilities, as well as hard to acquire kernel programming skills. With **Ipanema**, we propose a DSL-based approach that removes this last requirement. With its high-level abstractions, it also simplifies the translation of the scheduling algorithm to **safe** executable code usable in production environments. Additionally, our language enables to semi-automatically formally verify algorithmic properties.

We believe that Ipanema could be leveraged to ease the development of application-specific scheduling policies in existing OSs. We also believe that it could be a useful teaching tool in OSs courses.

FUTURE WORK. In the current state of our Ipanema tool chain, there are two axes for future contributions. The first one is to extend our proof library in order to verify new properties such as thread liveness. The second axis is to faithfully write existing schedulers in Ipanema and formally verify scheduling properties. This could lead to the discovery of bugs in existing schedulers such as CFS and attempts to fix them. More specifically, the latter axis would also lead to add new features to the Ipanema DSL. For example, we could add new data structures to store threads in addition to red-black trees and FIFO queues. We could also add more depth to the core states in order to manage other architectural attributes such as frequency or

temperature. This would add more flexibility to our language and allow for the implementation of more policies, existing or novel.

On top of Ipanema, in `SaaKM`, we could also work on a meta-scheduler to arbitrate between policies. For now, we only implement a static priority list. In the future, we wish to enable developers to write meta-schedulers that would schedule schedulers. This could be performed through the Ipanema DSL with new abstractions.

SUBMISSION TO THE LINUX COMMUNITY. Our work on the `SaaKM` internal API was not submitted to the Linux kernel community. It is not for lack of desire to do so, but because we know it will never be merged.

There was work on schedulers that tried to be integrated in the mainline code, but never achieved to do so. The reason behind this is that Linus Torvalds refuses to have multiple schedulers in Linux. The rationale is that scheduling is not that hard, so one scheduler should be enough. Additionally, having multiple schedulers means that improvements in one scheduler might never be merged into the other schedulers, and the development effort would be scattered across schedulers.

During the discussion on the (failed) merging of Con Kolivas' work in 2007, some developers raised the idea of having pluggable schedulers, like with `SaaKM`. Here is Linus Torvalds' answer:

> *I absolutely detest pluggable schedulers. They have a huge downside: they allow people to think that it's ok to make special-case schedulers. And I simply very fundamentally disagree.*
>
> *If you want to play with a scheduler of your own, go wild. It's easy (well, you'll find out that getting good results isn't, but that's a different thing). But actual pluggable schedulers just cause people to think that "oh, the scheduler performs badly under circumstance X, so let's tell people to use special scheduler Y for that case".*
>
> *And CPU scheduling really isn't that complicated. It's way simpler than IO scheduling. There simply is no excuse for not trying to do it well enough for all cases, or for having special-case stuff.*
>
> — Linus Torvalds, 2007 [174]

It is clear to us that there is no way our code will ever be merged.

RELATED PUBLICATIONS. The work described in this chapter has been the subject of two publications in international conferences [105, 106] and one in a national conference [67].

4

FREQUENCY-INFORMED SCHEDULING DECISIONS

In the previous chapter, we presented Ipanema, a DSL that enables scheduler developers to avoid safety bugs and formally verify scheduling properties. In addition to these potential bugs, schedulers are subject to another type of problem, **performance bugs**. These silent bugs eat away at performance but do not crash the machine. They are thus difficult to notice, except by accident while modifying the scheduler and obtaining better performance, or because of the performance of a competing scheduler. In Linux, this happened with the release of BFS that triggered changes in CFS that improved interactivity [95].

In this chapter, we propose multiple monitoring and visualization tools that will help developers detect, identify and solve such performance bugs. We will illustrate our methodology through an example: the discovery, study and resolution of the **frequency inversion** problem in CFS.

4.1 EXAMPLE OF A PERFORMANCE BUG IN CFS

While developing the Ipanema language presented in Section 3.2, we implemented multiple scheduling policies for testing purposes. One of them was the $local_{ipa}$ policy. Its design was simple: *(i)* for time-sharing and load balancing, behave in the same fashion as CFS albeit simplified; *(ii)* for thread placement, always place the waking (or new) thread where it (or its parent) previously resided. This policy was supposed to have poor performance due to its blatant violation of the work conservation property: using a single core instead of multiple ones is intuitively a bad idea. However, while running benchmarks, we discovered that reality was not that simple. Indeed, some applications, leader among them the compilation of the Linux kernel, proved to be executed faster with the $local_{ipa}$ policy than with CFS.

In order to understand why the $local_{ipa}$ policy outperforms CFS, we need tools to monitor the operations of the scheduler. These tools should help us understand the differences between both schedulers when executing this compilation. With this knowledge, we should be able to find out how to improve CFS and match the performance of $local_{ipa}$.

4.2 MONITORING AND VISUALIZATION TOOLS

Monitoring the operations of the Linux scheduler is a complicated task. First, the monitoring should be performed in the kernel since the monitored code resides there. Second, it must avoid the probe effect at all costs. This is important not only because of the time taken to execute the monitoring code, but also because the execution of this code could alter the decisions of the scheduler, rendering the monitoring useless.

Finally, users should be able to easily interpret the results. This requires visualization tools that are able to handle a large number of events efficiently. These tools should also provide customization options in order to allow users to look for specific patterns of events or specific data.

After a quick presentation of existing tools, we propose new tools that helped us better understand the behavior of schedulers. More specifically, our tools allowed us to discover a new performance bug in Linux, frequency inversion, and to devise new strategies in CFS to fix the problem.

4.2.1 *Existing Tools*

Before introducing our new tools, we first present existing tools and their pros and cons. These tools can be classified into two categories, *tracing* and *visualization* tools. The former are used to collect data while the latter are used to display the collected data to users.

4.2.1.1 *Tracing Tools*

The Linux kernel provides multiple tracing facilities to understand the behavior of the kernel. The `procfs` pseudo file system exposes kernel space data to user space through files that can be read from. For example, the `/proc/schedstat` file contains, for each CPU, information such as the number of times the `yield` system call was executed, the number of context switches or the time spent waiting by tasks. Such files are used by various user space utilities such as `htop` [126] or `ps` [142]. The information usually available in these files is either aggregate values or snapshots of the current state of the system. They cannot be used to trace all the operations of the scheduler.

The `ftrace` facility allows users to trace specific events in the kernel with minimal overhead. These events, called *tracepoints*, are hard-coded in the kernel source code. In the scheduler, there are 24 events as of Linux v5.7, including `sched_process_fork` or `sched_wakeup`. This facility can be controlled from user space through files located in the `/sys/kernel/debug/tracing/` directory or with the `trace-cmd` command line utility [153]. In addition, `ftrace` can be used to record the number of calls and the time spent in any function of the kernel. It

can also generate call graphs that are useful to find out how a function is called at run time.

The `perf` utility [137] can also be used to monitor the operations in the kernel. In addition to the support of tracepoints, `perf` supports software events such as page faults and hardware events such as cache misses.

Finally, the *eBPF* virtual machine embedded in the kernel can be used to trace the kernel [59]. Users can write probes that are compiled to *eBPF* byte code and inserted in the kernel at run time. These probes can be inserted at various predefined places in the kernel code such as tracepoints, hardware events or before or after any function call.[41] A set of tools using *eBPF*, `bpftrace` tools, is available and covers a variety of use cases, from measuring I/O latency to scheduling latency [24].

[41] There are some exceptions that are tagged as `notrace` in the kernel code.

4.2.1.2 *Visualization Tools*

When traces are recorded with the aforementioned tools, they still need to be presented in a way that eases the discovery of performance problems. Some of the tracing tools offer aggregate values that sum up results or human-readable interfaces, text or graphical. The `ftrace` and `perf` facilities provide the logs of events in a formatted text output, while `procfs` files usually offer aggregate values in text format.

KernelShark [151] is the official graphical viewer of traces generated by `ftrace`. It has a powerful filtering engine that helps visualizing only relevant events. SchedViz [65] is a tool developed by Google that also uses the output of `ftrace` but is more focused on the scheduler. It offers scheduler related views that expose problems due to NUMA or antagonisms between threads [64]. Trace Compass [53] is a graphical viewer that supports a large number of trace formats from kernel space, e. g. `ftrace` or `perf`, and from user space, e. g. `gdb` or user-defined traces. This enables developers to trace both sides of the system and correlate application-level events and kernel-level events.

4.2.2 *Our Tools*

In terms of tracing events related to the scheduler, we choose to use the existing `ftrace` facility. There are two main reasons for this choice: the negligible overhead and the ease of use. Indeed, `ftrace` has the smallest overhead among existing solutions. We also tried to implement a minimal event tracer in the kernel. The limited number of features allowed for efficient operations with minimal critical sections. Our implementation and `ftrace` performed similarly in terms of overhead. We therefore preferred `ftrace` since it provides more features and it is an official component of Linux, with various tools developed for it.

In terms of visualization, we were not satisfied by existing tools for two reasons. First, they were difficult to customize, making it difficult or even impossible to add new types of visualizations. Second, no tool

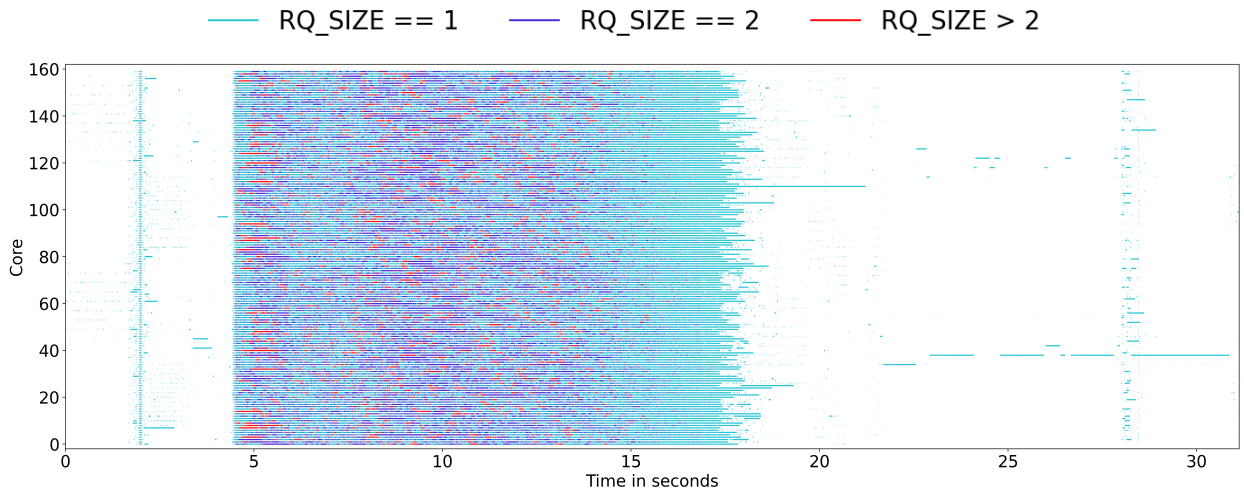


Figure 4.1: Size of the runqueues during a kernel compilation.

scaled correctly when handling traces from a big server machine with 160 cores. Existing tools either crashed or took way too long to display the traces because of the enormous number of events.

We therefore set out to develop a new tool that is programmable and scales to large machines. We propose **SchedDisplay**, a tool with a Python API that allows developers to easily change the data displayed and add new features. In order for our tool to scale to large traces, we only treat data that is in the displayed area and rely on the Bokeh library [21] for interactive visualization and the Datashader library [45] for rendering. Traces can also be exported in `svg` or `pdf` format, as is the case of the figures in this chapter.

We now present three examples of visualizations we implement in SchedDisplay to better understand the behavior of schedulers. The first one traces the size of the runqueues, the second one monitors the frequency of the CPU and the third one displays scheduler-related events, e. g. thread creations, wakeups, ...

4.2.2.1 *Runqueue Size*

This first tool is inspired by the one developed by Lozi *et al.* [113] and traces the number of threads in each runqueue. Instead of using a custom tracing facility as they did, we add a new tracepoint in the kernel. This tracepoint is triggered every time a thread is enqueued or dequeued from a runqueue and reports the new size of the runqueue.

Figure 4.1 shows an example of such a trace. The X-axis represents time while each line on the Y-axis represents a core. If the core has no thread in its runqueue, i. e. the core is idle, no line is drawn. The line is blue if a single thread is present on the core, purple if two threads are present and red if more threads are present. The tool can easily be customized to change the ranges of runqueue sizes and to

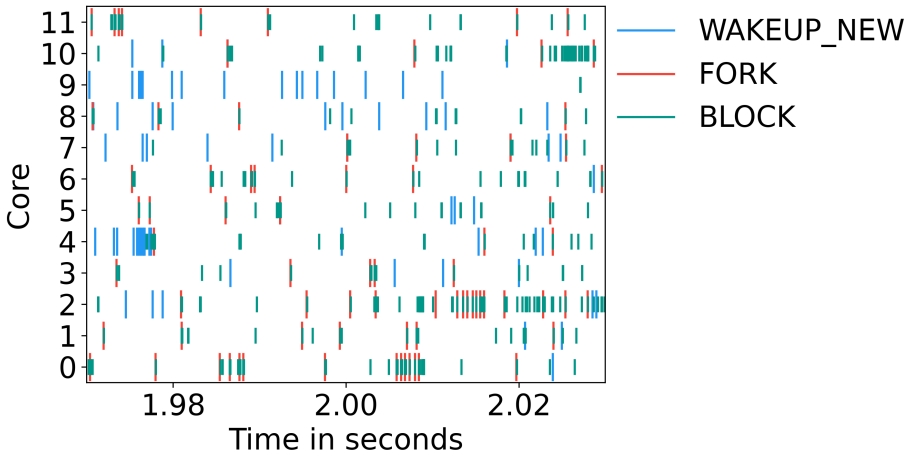


Figure 4.2: Example of the scheduler events visualization.

select which are displayed. With such a visualization, we can easily discern the phases of an application where the machine is overloaded or underused.

4.2.2.2 *Monitoring the Frequency*

The previous tool assumes that all cores are equivalent. Two threads on core 1 is the same as two threads on core 12. This is, however, not entirely true on modern CPUs. With frequency scaling technologies, two cores might have a different computing power at a given time. If core 1 is running at 1 GHz and core 12 at 4 GHz, the two threads on core 12 will execute faster than those on core 1.

To monitor this, we add a new tracepoint, `sched_tick`, that records every clock tick in the system. In this tracepoint, we record the currently running thread's PID, the flag that indicates if a new election should be performed and the current frequency of the core. Figure 4.3 shows the evolution of the frequency of a 160-core machine during a compilation. We will come back to this example with more details in Section 4.3.1.

4.2.2.3 *Scheduler Events*

Finally, our last tool allows us to better understand the behavior of the scheduler itself by displaying scheduler-related events. These events are collected through tracepoints and include thread creations, blocking events or wakeups. Figure 4.2 shows an example of such a trace. With this representation, we can find sequences of related events that are problematic. This will be particularly useful to find the root cause of the problem we discover.

In addition to events directly related to the scheduler, we can visualize any tracepoint recorded with `ftrace`. We can therefore use this tool to make correlations with other parts of the kernel such as

synchronization mechanisms or any system call. Additionally, we can also use this facility to trace events from our `SaaKM` scheduling class presented in Chapter 3.

4.3 INVESTIGATING THE PERFORMANCE BUG

4.3.1 *Preliminary Investigation*

We start our investigation by running the compilation of the Linux kernel while recording scheduler events with `ftrace`. Figure 4.3 shows the trace of this compilation with 320 jobs on a 4-socket 160-core Intel® Xeon E7–8870 v4 server with CFS and $local_{ipa}$. The X-axis represents time while each line on the Y-axis represents a core. This figure depicts the frequency of the core recorded at each tick.

EXECUTION WITH CFS. First, let's detail the behavior of this compilation with CFS on Figure 4.3a. We observe that the application can be split into three phases A, B and C, as outlined in the figure.

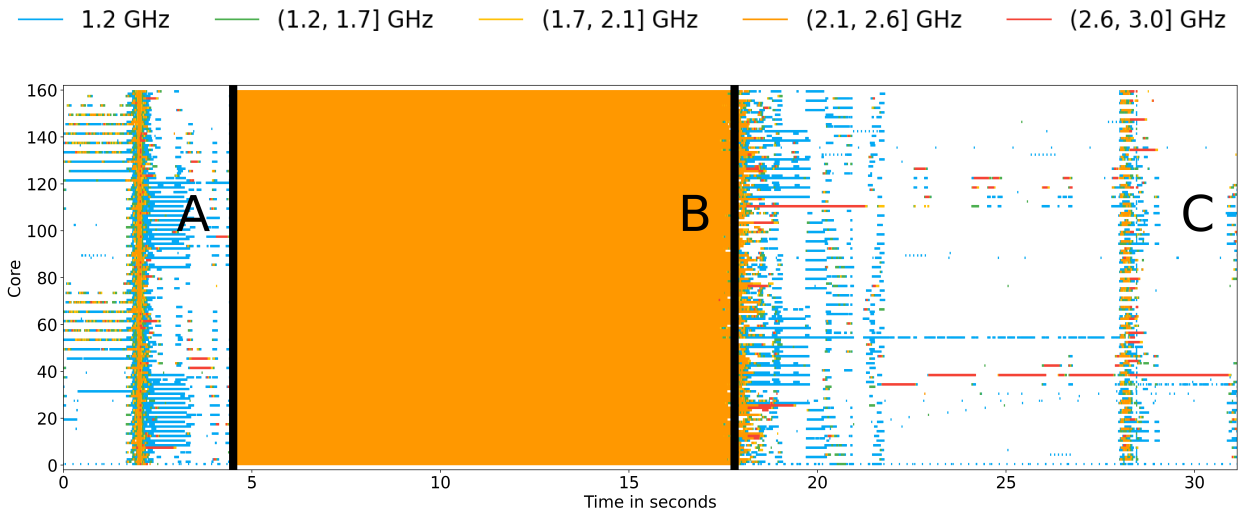
Phase A corresponds to preparatory scripts that parse the configuration options of the kernel and generate headers and other files needed for the compilation. This phase is not very parallel, with fewer than ten cores running concurrently most of the time. When cores are not idle (white), they mostly run at a low frequency (less than 2.1 GHz). Note that there is a quick parallel burst around 2 seconds when all cores run at a high frequency.

Phase B corresponds to the compilation of files from their textual representation (C language) to binary language (assembly). Each file is compiled independently, making this phase highly parallel. The machine is overloaded and all cores run at a high frequency, between 2.1 GHz and 2.6 GHz.

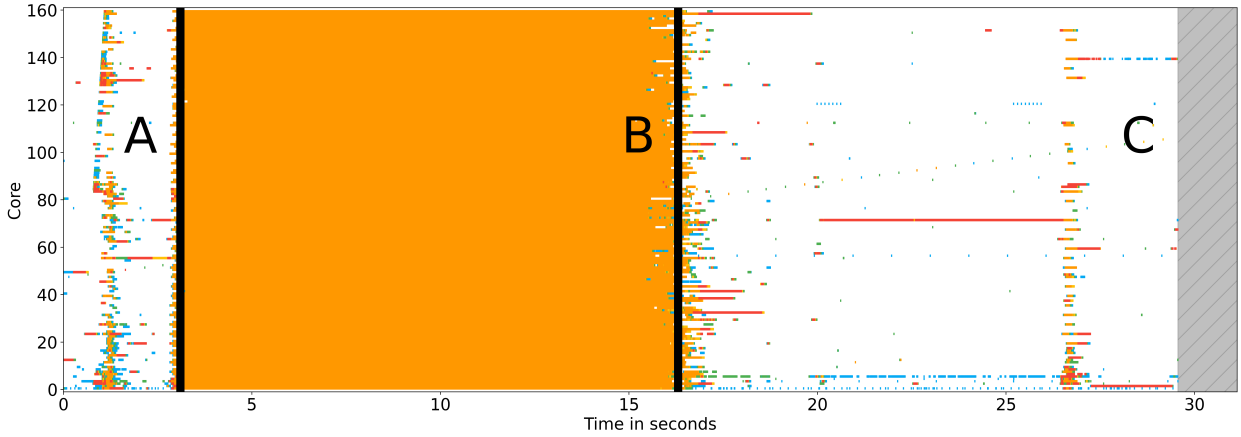
Finally, phase C corresponds to linkage and generation of the final binary file. The behavior is close to phase A, with only a few cores used. Again, we see a short burst of activity at the end. This burst is due to the compilation of modules.

EXECUTION WITH $local_{ipa}$. Now, let's see how this compilation is different with the $local_{ipa}$ scheduler on Figure 4.3b. Note that the X-axis is at scale with the previous figure, the grayed out area corresponds to the time after the completion of the compilation.

We still distinctly see the three phases A, B and C. However, phase A is faster with this scheduler. We can observe that fewer cores are usually running compared to CFS albeit at a higher frequency. This is due to the Intel® Turbo Boost technology that allows some cores to run at very high frequencies if other cores are idle. This idleness is expected since this scheduler always places threads locally and violates the work conservation property. Threads are spread only



(a) With CFS



(b) With $local_{ipa}$

Figure 4.3: Execution trace of the compilation of the Linux kernel with 320 jobs on our 160-core Intel® Xeon E7–8870 v4 server.

when idle or periodic load balancing is performed, which happens rarely. As with CFS, phase C is similar to phase A.

Phase B behaves exactly like CFS and completes with the same duration. Since the machine is overloaded (320 jobs for 160 cores), load balancing is efficient and always finds threads to steal, minimizing idleness.

CONCLUSIONS. In this benchmark, $local_{ipa}$ outperforms CFS in the phases when the machine is underutilized, i. e. A and C. In these phases, $local_{ipa}$ seems to take advantage of higher core frequencies than CFS by using fewer cores. The question that arises is why cores run at such low frequencies with CFS even though only a few threads are running. To find out what the reason behind this behavior is, we

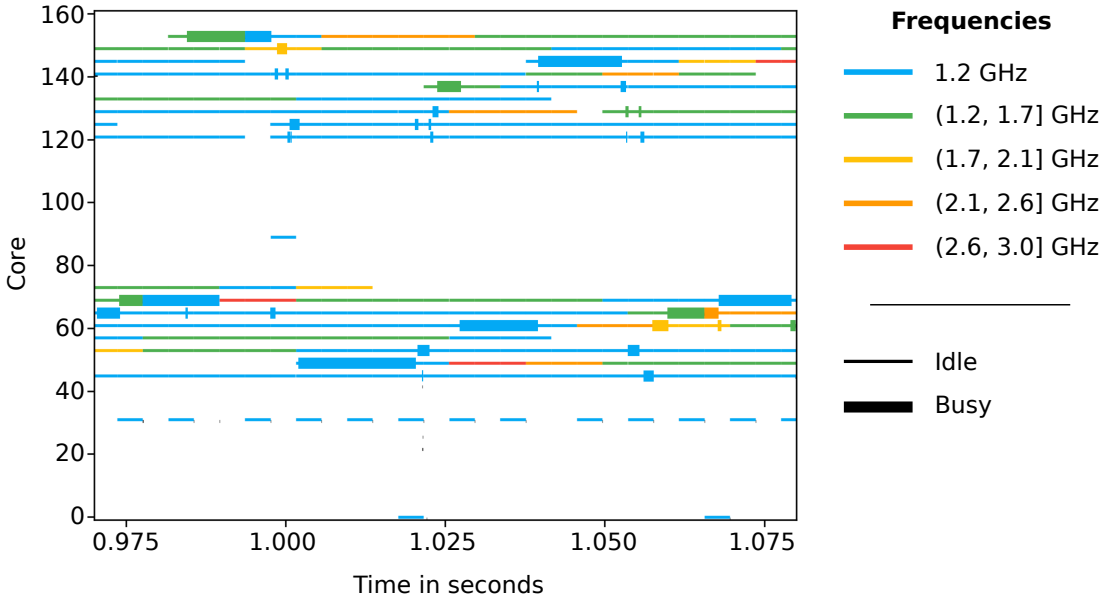


Figure 4.4: Zoom over a 100 ms period of phase A with CFS.

concentrate on phase A and analyze the scheduling events that lead to this inefficiency.

4.3.2 *The Frequency Inversion Problem*

The traces we collect contain a large number of events related to scheduling, making it difficult to skim through all this information. To ease this process, we focus on a small window of time at the beginning of the experiment, during phase A. By doing this, we will pinpoint the exact problem behind the performance issue of CFS and the origin of this problem.

THE PROBLEM. Figure 4.4 shows the same frequency data as previously, but only on a 100 ms period. In addition to the frequency, the thickness of the line represents the idleness of the core. If the line is thin, no thread is runnable on this core, i. e. the core is idle. If the line is thick, at least one thread is runnable on this core, i. e. the core is busy.[42] For example, just after the 1.000 s mark, a thread is placed on core 50. This core remains busy until just before the 1.025 s mark. It then stays idle until the end of the displayed period of time.

On this figure, we notice that every time a core becomes busy, i. e. the line becomes thicker, the core is running at a low frequency. This makes sense since an idle core should have a low frequency to save energy. However, we also notice that cores become idle again before the frequency is raised to a high value. The frequency attains its maximum value when the core is idle again. This change of frequency happens too late to be useful.

[42] This representation is a combination of the runqueue size and frequency visualization produced thanks to our scriptable tool.

Another peculiarity is observable in this figure: there seems to never be two threads running concurrently. This means that the execution that happens during this window of time is actually sequential. However, instead of using only a single core, which would have been enough, CFS uses more than a dozen cores.

The combination of these two phenomena triggers our problem. Idle cores are used to run a thread, and just before the frequency reaches a high value, the computation is moved to another idle and low frequency core, making the previous one idle again, but at a high frequency. Consequently, the frequencies at which both cores operate are *inverted* as compared the actual load on the cores. We call this problem **frequency inversion**.

SOURCE OF FREQUENCY INVERSIONS. We have now identified the problem from which CFS suffers. In order to fix this problem, we must find its origin. To this end, we study the scheduler-related events that happen during the studied window of time. Figure 4.5 shows a subset of the events happening during this period. We display only three events for clarity:

- `FORK`: The `fork()/clone()` system call is being called on this core, i. e. the parent's core,
- `WAKEUP NEW`: A newly created thread is being placed on this core, i. e. child's core, following a `FORK` event,
- `BLOCK`: A thread is going into a BLOCKED state.

We also display the idleness of cores on the figure. When a black line is present, the core is busy, i. e. the core has at least one runnable thread. Since we show the same time window as in Figure 4.4, black lines map the thick lines of the previous figure.

On the frequency trace, we already figured out that this part of the execution is sequential but still runs on multiple cores. We now analyze the events that happen when these changes of core take place. We can see that each time there is a black line break, i. e. a migration of the work from one core to another, there is a `fork()` followed by a blocking event on the first core. This is a classic `fork()/wait()` pattern used in many applications. Here, it is the `make` program that spawns a thread per job to perform and waits for them to complete. Since this phase is sequential, only one thread at a time is working.

When a thread is created with the `fork()` system call, CFS searches for the least loaded core to place this new thread on it. In this under-utilized machine, this corresponds to an idle core. When this child thread is placed on its core, the parent immediately calls the `wait()` system call, blocks and leaves its core idle.

This behavior would not be a problem if the frequency was able to immediately scale to the actual load on the core. However, we

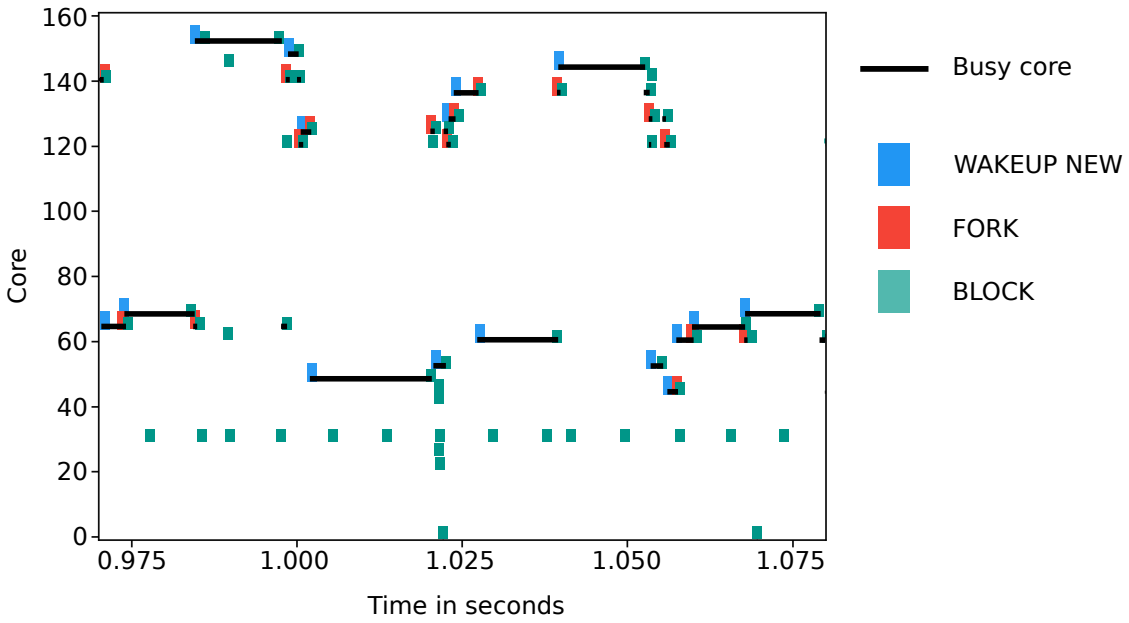


Figure 4.5: Scheduling events over the 100 ms period of phase A with CFS.

previously saw that this was not the case. CFS acts as if all cores are equal, but this becomes less and less true, even on homogeneous architectures because of frequency scaling.

HOW DO WE SOLVE THIS PROBLEM? With the frequency inversion problem identified, we need to find a solution. In order to be able to do so, we first need to better understand how dynamic frequency scaling works. With this information, we will be able to devise the best strategy to adopt to mitigate frequency inversions.

4.4 DYNAMIC FREQUENCY SCALING

On modern processors, the frequency can dynamically change in order to increase performance or save energy. The former is straightforward: a higher frequency means that more instructions can be executed per second. The latter comes from the relation between energy consumption and frequency:

$$P = C \times V^2 \times f$$

where P is the power consumed, C is the capacitance of the circuit (a constant for a given chip), V is the operational voltage and f is the operational frequency.[43]

[43] This formula can be inferred from Ohm's law.

There is a linear relation between frequency and power consumption. Dividing the frequency by two automatically divides the energy consumed by two. In addition to this relation, voltage and frequency are also closely related. Higher frequencies require a higher voltage, while low frequencies require a lower voltage. Combined with the

quadratic relationship between power consumption and voltage, reducing frequency is a good way of lowering the energy consumed by the CPU.

Most CPU vendors implement dynamic voltage and frequency scaling (DVFS) mechanisms. We will present implementations from multiple vendors and how Linux leverages DVFS. Finally, we will evaluate the frequency transition latency (FTL) on multiple CPUs from Intel® and AMD®.

4.4.1 *Frequency-related Technologies on Intel® Platforms*

Since the early 2000s, Intel® introduced multiple technologies related to the operating frequency of its processors. The Enhanced Intel Speed-Step Technology (EIST) [82], introduced in 2005, allows the software to change the frequency of a whole processor chip dynamically through Performance States (or P-States) levels. Each P-State level is associated with a given voltage and frequency range. This means that the software (e. g. the OS) can choose a P-State depending on various metrics such as the load of the system, or whether the user wants the highest possible performance, regardless of the energy usage. In Linux, the subsystem responsible for this control is called `CPUFreq` (we will provide more details about this in Section 4.4.3).

In 2015, Intel® introduced the Speed Shift [78] technology that gives frequency and voltage control back to the hardware. With this technology, the OS can choose to let the processor decide by itself the best target voltage and frequency for the current workload. This allows for faster and more precise frequency changes, with P-States being replaced by a non-discrete range of possible frequencies. As with EIST, frequency is managed at the chip granularity.

Finally, in 2019, with its Cascade Lake microarchitecture, Intel® announced the Speed Select Technology [47] that enables per-core frequency control instead of per-chip control. However, even though this feature was only announced for the new Cascade Lake processors, we observed different per-core frequencies on an older Xeon processor from the Broadwell generation released in 2016. This feature might have already existed on processors targeted for servers without being marketed by Intel®.

Two other technologies boost the maximum performance of some cores at the expanse of others: Turbo Boost [83] and Thermal Velocity Boost [180]. Turbo Boost increases the frequency of a subset of cores beyond their base frequency for a short period of time if other cores are inactive. Thermal Velocity Boost gives another small boost over Turbo Boost if the temperature of the computer case is low.

4.4.2 *Frequency-related Technologies of Other Vendors*

AMD. Like Intel®, AMD® implements technologies to boost the maximal frequency of a CPU when possible. Turbo Core [5] and Precision Boost [6] technologies allow cores to exceed their normal frequency depending on the number of active cores, temperature and power consumption. All AMD® CPUs since the introduction of the Zen microarchitecture in 2017 feature per-core dynamic frequency scaling.

In terms of frequency control, AMD® also features P-states that allow the OS to dynamically choose the frequency depending on the workload and on the user-defined policy.

ARM. ARM® also implements a DVFS mechanism controlled by the OS through Operating Performance Points (OPPs). OPPs are similar to P-states: they are a tuple containing a frequency and a voltage. The set of available OPPs is implementation-specific.

In terms of DVFS granularity, it seems that frequency is managed at the chip level. Even though we did not find clear evidence in the official documentation of this, we infer this information from various documentation on how to use the `CPUFreq` interface of Linux on ARM® chips. These documentation usually instruct to modify the policy of core 0 in order to change the policy of all cores. However, we found a marketing document for a Snapdragon chip using the ARMv7 architecture that specifically mentions per-core DVFS [144]. Per-core DVFS might therefore be an implementation-specific feature.

Note that we distinguish per-core DVFS from the heterogeneous big.LITTLE architecture found on many ARM® chips. In this type of chips, the different frequencies are due to the different kind of cores present, not to a specific per-core management of DVFS.

4.4.3 *Dynamic Frequency Scaling in the Linux Kernel*

In the Linux kernel, dynamic voltage and frequency scaling (DVFS) is managed by the `CPUFreq` [183] subsystem with two mechanisms: the *CPU scaling governor* and the *CPU scaling driver*. The former determines the targeted frequency and voltage of the processor, and the latter is an architecture-specific component that interacts with the hardware to enact the governor's decision.

4.4.3.1 *CPU Scaling Driver*

The CPU scaling driver is the architecture-specific component of the OS that is responsible for changing the processor's frequency and voltage. Depending on the hardware, this is done at the chip or at the core level. Such drivers must implement a set of handlers called by other subsystems. Such handlers include getting or setting the current frequency, suspending the CPU, or activating boost frequencies.

Usually, these drivers control the hardware by reading or writing to hardware registers. These drivers are only responsible for the communication between the kernel and the hardware, they are not supposed to determine the frequency that should be set.

4.4.3.2 *CPU Scaling Governor*

The CPU scaling governor is the component that determines which frequency should be used. It holds multiple algorithms that input system information, e. g. CPU load, and hardware capabilities, i. e. through the CPU scaling driver, and outputs a desired frequency. The Linux kernel provides generic governors, each having a different goal.

The `performance` governor always requests the maximum frequency available[44] when enabled and does not dynamically change the frequency. The goal of this governor is to maximize performance regardless of the energy expended and the system load.

[44] Hardware-defined by default, but a lower limit can be set by the user.

The `powersave` governor, on the other hand, always requests the lowest possible frequency when enabled, and does not dynamically change the frequency either. The goal of this governor is to save energy at the expense of performance.

The `ondemand` governor dynamically changes the frequency proportional to the system load. The system load is estimated as the proportion of time the CPU was not idle. This estimate is recomputed periodically. If the estimated system load is 75%, the selected frequency will be 75% of the range between the maximum and minimum frequencies. The goal of this governor is to provide good performance when needed and save energy otherwise.

Finally, the `schedutil` governor dynamically changes the frequency depending on the CPU load estimated by the thread scheduler. When running a real-time thread, i. e. using the `rt` or `dl` scheduling classes, the frequency will be set to the maximum value. Otherwise, the next frequency F_{next} is computed with the formula:

$$F_{next} = 1.25 \times F_{curr} \times \frac{L_{curr}}{L_{max}}$$

where F_{curr} is the current frequency, L_{curr} is the current load computed by CFS and L_{max} is the maximum possible load. The goal is the same as for the `ondemand` governor, but the metric that quantifies the load of the processor is supposed to be better since it is derived from more precise data coming from the scheduler.

These governors are generic implementations that work on any underlying scaling driver. However, some scaling drivers redefine the behavior of some governors in order to use certain hardware features. For example, Intel® redefines the `powersave` governor in its `intel_pstate` driver [184]. It no longer statically sets the lower frequency, and instead lets the hardware decide the frequency that should be used. On modern Intel® processors, the hardware determines the

current CPU usage and sets a frequency accordingly, in the same vein as the `ondemand` governor.

4.4.4 *Limits of Hardware Reconfiguration*

All these hardware technologies configure the hardware (voltage, frequency) in order to match the software requirements. However, our case study, the compilation of the Linux kernel, shows that this is not sufficient to make the best of our hardware. Indeed, these hardware reconfigurations are not instantaneous and lead to bad processing power usage by the OS (especially the thread scheduler). We present a characterization of the reconfiguration delays by measuring the frequency transition latency (FTL), and study this latency on multiple CPUs.

4.4.4.1 *Frequency Transition Latency*

Relying on hardware reconfiguration technologies such as EIST means that, ideally, the frequency of all cores should be the best one possible at any given time. This implies that reconfiguration should instantaneously mirror the load of a core. Mazouz *et al.* [117] studied the time Intel® processors took to change their frequency. They measure the hardware delay between the command and the actual change of frequency. The delay depends on the source and destination frequencies, but they all are in the order of tens of microseconds. Yet, our previous frequency traces seem to show that this is underestimated. Therefore, we set to measure the FTLs of our hardware.

Mazouz *et al.* define the FTL as the duration a core takes to go from one frequency to another. In this work, we define the FTL a little bit differently. In addition to the hardware transition latency, we include the decision-making process in the FTL. We wish to measure the time it takes for a core to reflect the software load on the frequency. This means that decisions taken by the scaling governor are also measured. We also differentiate two types of latency: the rising latency FTL_R and the decreasing latency FTL_D. FTL_R is the time a core takes to go from its lowest to its highest frequency, while FTL_D is the converse. This is important because one might be different from the other depending on the goals of the governor, i. e. energy versus performance.

4.4.4.2 *Measuring the Frequency Transition Latency*

In order to measure the FTL on real hardware, we need precise frequency monitoring tools. We develop `frequency_logger`[1], that monitors the frequency of a given core at a given interval of time. The frequency ($\text{freq}_{current}$) is computed by reading two model-specific reg-

1 Available at: https://github.com/rgouicem/frequency_logger

isters (MSRs), APERF and MPERF, and multiplying their ratio with the processor's base frequency ($freq_{base}$):

$$freq_{current} = \frac{APERF}{MPERF} \times freq_{base}$$

Each core has its own set of MSRs. The APERF register continuously increments in proportion to the current frequency while the MPERF register continuously increments in proportion to a fixed frequency, hence the previous formula. The timestamps are recorded with the `rdtsc` x86 instruction that returns the number of elapsed cycles since boot at a fixed frequency, which can then be converted to nanoseconds.

In order to ensure that the measurement has no side effect on the experiment, the thread running `frequency_logger` should not run on the monitored core. Additionally, we make sure that the monitored and monitoring cores do not share computing hardware (SMT), since the activity of the monitoring thread might raise the frequency of the monitored thread.

To measure the FTL, we run the following microbenchmark, depicted in Figure 4.6:

1. Run `frequency_logger` pinned on core 0, set up to monitor core 1 every millisecond,
2. Sleep B ms on core 1,
3. Run an infinite loop on core 1 for D ms,
4. Sleep A ms on core 1 again.

The values of B, D and A depend on the tested processor. We choose the smallest possible values while allowing the processor to meet the maximum frequency in the rising phase and the minimum frequency in the decreasing phase. This experiment exhibits two FTLs: a rising FTL (FTL_R) accounting for the duration between the start of the loop and the time the core attains its maximum frequency, and a decreasing FTL (FTL_D) accounting for the duration between the end of the loop and the time the core attains its lowest frequency stably.

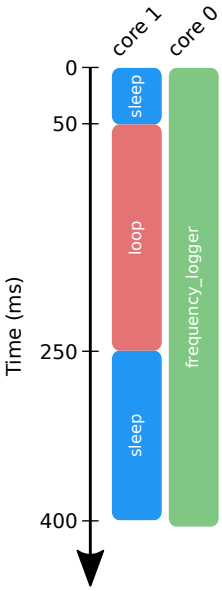


Figure 4.6: Execution of `frequency_logger`.

4.4.4.3 *Study of Frequency Transition Latencies*

We run our microbenchmark on multiple machines and report the results in Table 4.1. Our machines include servers, desktops and laptops, with mostly Intel® CPUs. We run these experiments with the `powersave` governor for all Intel® CPUs and with the `schedutil` governor for the AMD® CPU. This choice is due to the implemented policies of these governors, as explained in Section 4.4.3.2. For this FTL measure to be meaningful, we need a governor that will dynamically change the frequency to meet the software requirements. The `powersave` governor on Intel® processors does so, as does the

			Frequency (GHz)			FTL (ms)	
Market	**CPU Model**	**Cores**	Base	Min	Max	FTL_R	FTL_D
Server	Intel® Xeon E7–8870 v4	160	2.1	1.2	3.0	29	98
Server	Intel® Xeon E5–2699 v4	88	2.2	1.2	3.6	378	69
Desktop	Intel® Core i7 8086K	12	4.0	0.8	5.0	26	81
Desktop	AMD® Ryzen 5 3400G	8	3.7	1.4	4.2	194	51
Laptop	Intel® Core i5 8350U	8	1.7	0.4	3.6	134	25
Laptop	Intel® Core i7 7500U	8	2.7	0.4	3.5	168	224

Table 4.1: Frequency transition latencies of multiple CPUs.

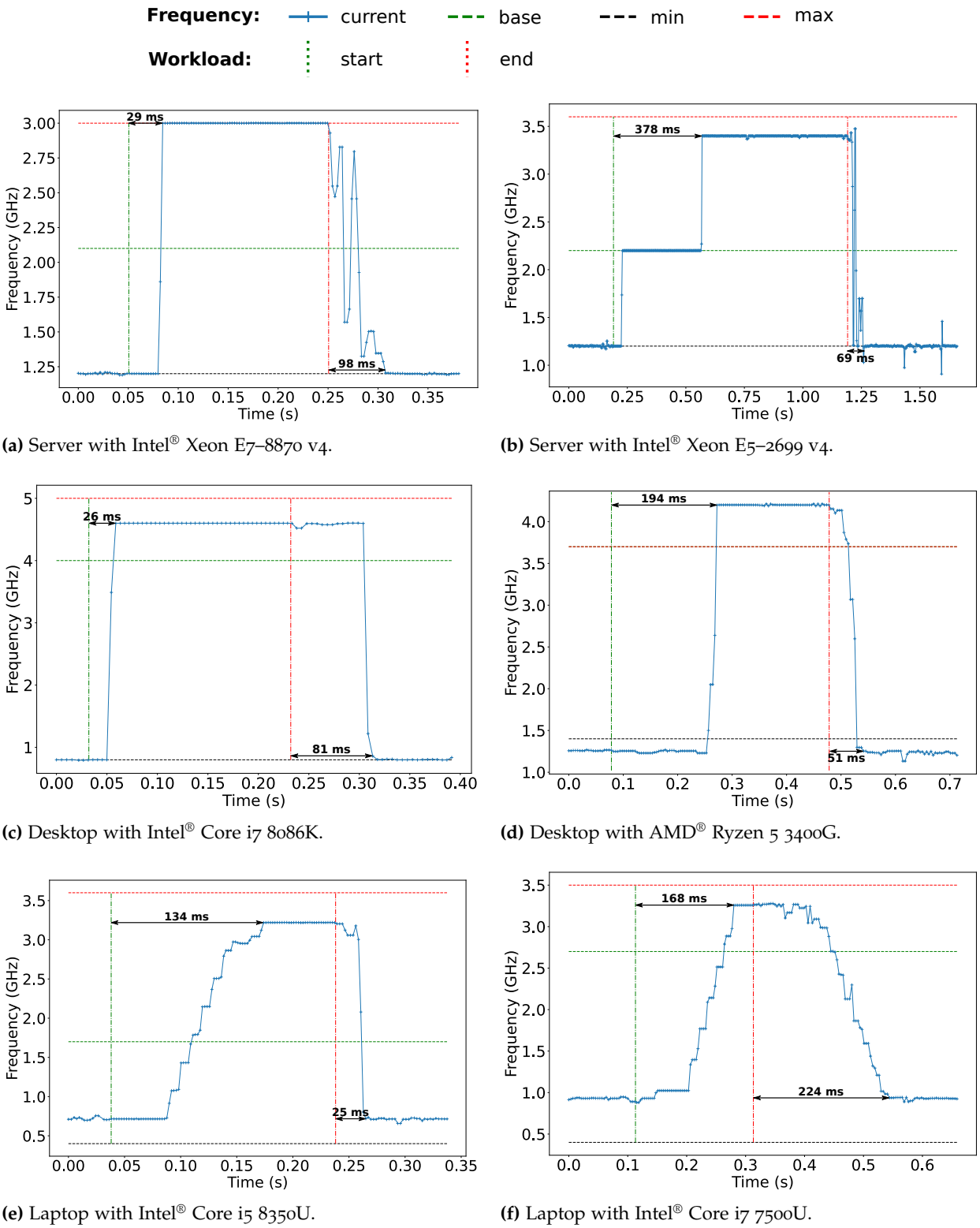
`schedutil` governor on all CPUs. However, the `powersave` governor produces shortest FTLs than the `schedutil` governor on our Intel® machines, hence the choice to use the former when possible.

Figure 4.7 shows the evolution of the frequency during the aforementioned scenario on every tested machine. Vertical lines indicate the start and the end of the busy loop. We can observe that each machine has a different behavior, depending on its targeted market, i. e. server, desktop or laptop, and its product line, i. e. entry level, mainstream or high-end.

ULTRA HIGH-END. Among our tested CPUs, two can be categorized as *ultra high-end* products, the Intel® Xeon E7–8870 v4 and the Intel® Core i7 8086K (Figures 4.7a and 4.7c). Since they are both targeted for high performance, they both have a very short FTL_R and minimize the time spent busy at a low frequency. They also have a fairly long FTL_D in order to still be at a high frequency if new work becomes available. These models favor performance over energy.

MAINSTREAM. We also have two *mainstream* processors, the AMD® Ryzen 5 3400G and the Intel® Core i5 8350U (Figures 4.7d and 4.7e). These products are more balanced than high-end models. The FTL_D is shorter than with high-end models and the FTL_R slightly longer. The frequency takes longer to match an increasing load and goes back to a lower power mode faster. Even though these processors are more balanced than high-end models, we note that the balance leans towards saving energy ($FTL_R > FTL_D$).

OUTLIERS. The Intel® Core i7 7500U processor is a *high-end* laptop CPU (Figure 4.7f). It is quite balanced in its frequency behavior, with similar FTL_D and FTL_R values, but has very high latencies. It still leans towards performance, as suggested by its high-end product



(a) Server with Intel® Xeon E7–8870 v4.

(b) Server with Intel® Xeon E5–2699 v4.

(c) Desktop with Intel® Core i7 8086K.

(d) Desktop with AMD® Ryzen 5 3400G.

(e) Laptop with Intel® Core i5 8350U.

(f) Laptop with Intel® Core i7 7500U.

Figure 4.7: Measures of frequency transition latencies.

placement. We are, however, not quite sure of the reason behind these long FTLs.

The Intel® Xeon E5–2699 v4 processor is a *high-end* server CPU (Figure 4.7b). This model has a particular behavior considering its placement amid other Intel® processors. Instead of going straight from minimum to maximum frequency, we observe a long plateau at the base frequency. It reaches the base frequency in 38 ms only, then maintains this frequency for 341 ms before using the highest possible frequency. Again, we do not have a convincing explanation for this behavior, especially when comparing it to other older and more recent Intel® CPUs.

4.5 HANDLING FREQUENCY INVERSIONS IN CFS

Now that we have precisely identified the frequency inversion problem, we develop new thread placement strategies that we implement in CFS. We propose two placement strategies: one that drastically changes the behavior of CFS, and another that only slightly changes CFS's behavior.

4.5.1 S_{local}: The Local Placement Strategy

Our first strategy is S_{local} and consists of implementing a similar placement strategy as our $local_{ipa}$ policy, but integrated in CFS. We modify CFS so that when a thread is created[45] or wakes up, it is placed on the same core as the thread that created it or woke it up. In the context of the `fork()/wait()` pattern described earlier, this strategy ensures that the created thread is more likely to run on a core operating at a high frequency. This is because the parent thread was active on the same core, increasing the likelihood of this core's frequency to have been increased already. Furthermore, since the parent thread calls the `wait()` system call shortly after calling `fork()`, both threads will share the same core for a limited period of time.

[45] A thread creation might be due to a `fork()` or a `clone()` system call.

In the context of a producer-consumer application, when a producer wakes a consumer, the S_{local} strategy will place the woken thread on the same core as its waker. And, as with the `fork()/wait()` pattern, the consumer will most likely be placed on a high-frequency core due to the activity of the producer thread. The time during which both threads will share the same core will also be limited because the producer is likely to block or terminate after waking a consumer. This could happen if the shared buffer is not big enough to allow both threads to run or if producing is faster than consuming (or the other way around).

Although this strategy seems to be beneficial for workloads resembling our case study, performance will be seriously hurt in other situations. For both patterns exhibited previously, if the parent thread

or the producer thread does not block or terminate promptly (with the `wait()` system call for example), the core where both threads reside will be overloaded for a certain period of time. The periodic load balancer of CFS mitigates this by eventually migrating one of these two threads to an idle core if any is available. However, waiting for the next periodic load balancing could be quite long. In CFS, load balancing is performed hierarchically, with varying periods depending on the level in the hierarchy: cores in the same cache domain are more frequently balanced than cores residing on different NUMA nodes. These periods vary from 4 to 320 milliseconds on our 4-socket NUMA server.

In terms of implementation, although this strategy significantly changes the behavior of CFS, it only required the modification of three lines of code in the scheduler. The patch for Linux v5.4 is available here: `https://gitlab.inria.fr/whisper-public/atc20`.

4.5.2 S_{move}: *The Deferred Thread Migration Strategy*

Our S_{local} strategy's major flaw is core oversubscription, even though it is mitigated by load balancing. To fix this flaw while keeping the benefits of local placement, we design the S_{move} strategy. In CFS, when a thread is created or woken, the scheduler decides on which core it should run and places it immediately. Our S_{move} strategy uncouples the decision and the migration in order to take advantage of a high-frequency core. To do so, we save the decision of CFS and defer the actual migration.

Let T_{wakee} be the newly created or waking up thread, T_{waker} the thread creating or waking up T_{wakee}, C_{waker} the core where T_{waker} is running and C_{CFS} the destination core chosen by CFS. The normal behavior of CFS is to directly enqueue T_{wakee} in the runqueue of C_{CFS}. In S_{move}, we delay this migration to allow T_{wakee} to run on a high-frequency core if C_{CFS} is running at a low frequency. If C_{CFS} is running at a frequency higher than the CPU's minimal frequency (i. e. the core is not idle frequency-wise), we enqueue T_{wakee} in C_{CFS}'s runqueue immediately. This is the default behavior of CFS. Otherwise, we arm a high-resolution timer interrupt that will perform this migration in D μs and enqueue T_{wakee} in C_{waker}'s runqueue. If T_{wakee} is scheduled on C_{waker} before the interrupt is triggered, we cancel the timer.

The rationale behind S_{move} is to avoid waking low-frequency cores if the task can be performed quickly when placed locally, i. e. on a core that is likely to run at a high frequency. Indeed, T_{waker} is running at the time of the decision, meaning that C_{waker} is likely to run at a high frequency.

The delay D can be changed at run time by writing to a special file in the `sysfs` pseudo-filesystem. We choose a default value of 50 μs, which is close to the delay between a `fork()` and a `wait()` system

call during our Linux kernel build experiment. Varying this delay D between 25 μs and 1 ms did not have significant impact on the benchmarks of Section 4.6. The frequency threshold used to determine if a core is idle frequency-wise (defaults to the minimal frequency of the CPU) can also be changed through the `sysfs` if this frequency is not appropriate for the CPU used.

In terms of implementation, the S_{move} strategy required to add or modify 124 lines of code in CFS. The patch for Linux v5.4 is available here: `https://gitlab.inria.fr/whisper-public/atc20`.

4.6 EVALUATION

In this section, we evaluate both S_{local} and S_{move} strategies against CFS. Our objective is to compare the performance and energy usage of both strategies and evaluate the improvements as compared to CFS.

We run a large number of applications from two established benchmark suites, Phoronix [119] and NAS [13], as well as other applications, such as `hackbench` (a popular benchmark in the Linux kernel scheduler community) and `sysbench OLTP` (a database benchmark).

We run these experiments on a server grade 4-socket NUMA machine with Intel® Xeon E7–8870 v4 processors and a desktop machine with an AMD® Ryzen 5 3400G CPU, as presented in Table 4.2.[46] S_{local} and S_{move} were both implemented in the Linux v5.4 kernel, and compared against the same unaltered version.

[46] Of the six machines presented in Table 4.1, these two are the only ones capable of per-core frequency scaling.

We run all experiments 10 times. We measure energy consumption with the running average power limit (RAPL) feature that exposes the consumed energy through hardware registers. The server machine exposes energy consumed by the CPU package and the dynamic random access memory (DRAM), while the desktop machine only exposes that of its CPU package consumption. We choose to include the DRAM consumption for the server machine in order to have the most precise value possible.

While the energy consumption is always presented in joules (J), the performance metric varies from one benchmark to another (latency, execution time, throughput), with inconsistent units. For better readability, all the following graphs show the improvement in terms of performance and energy compared to the mean of the 10 runs with CFS. Therefore, higher is always better, regardless of the unit. The raw value of the mean for CFS is displayed on the top of the graphs, alongside the benchmark's unit.

To demonstrate that our work is orthogonal to the governor used, we evaluate our strategies using both the `powersave` and the `schedutil` governors. As presented in Section 4.4.3.2, the `powersave` governor on Intel® machines lets the hardware decide which frequency should be used with regards to the current load. The `schedutil` governor is the latest governor designed by the Linux community that takes

	Server	Desktop
CPU model	Intel® Xeon E7–8870 v4	AMD® Ryzen 5 3400G
Cores (SMT)	80 (160)	4 (8)
NUMA nodes	4	1
Minimum frequency	1.2 GHz	1.4 GHz
Base frequency	2.1 GHz	3.7 GHz
Max Turbo frequency	3.0 GHz	4.2 GHz
Memory	512 GB	8 GB
OS	Debian 10 (buster)	Arch Linux

Table 4.2: Configurations of our experimental machines.

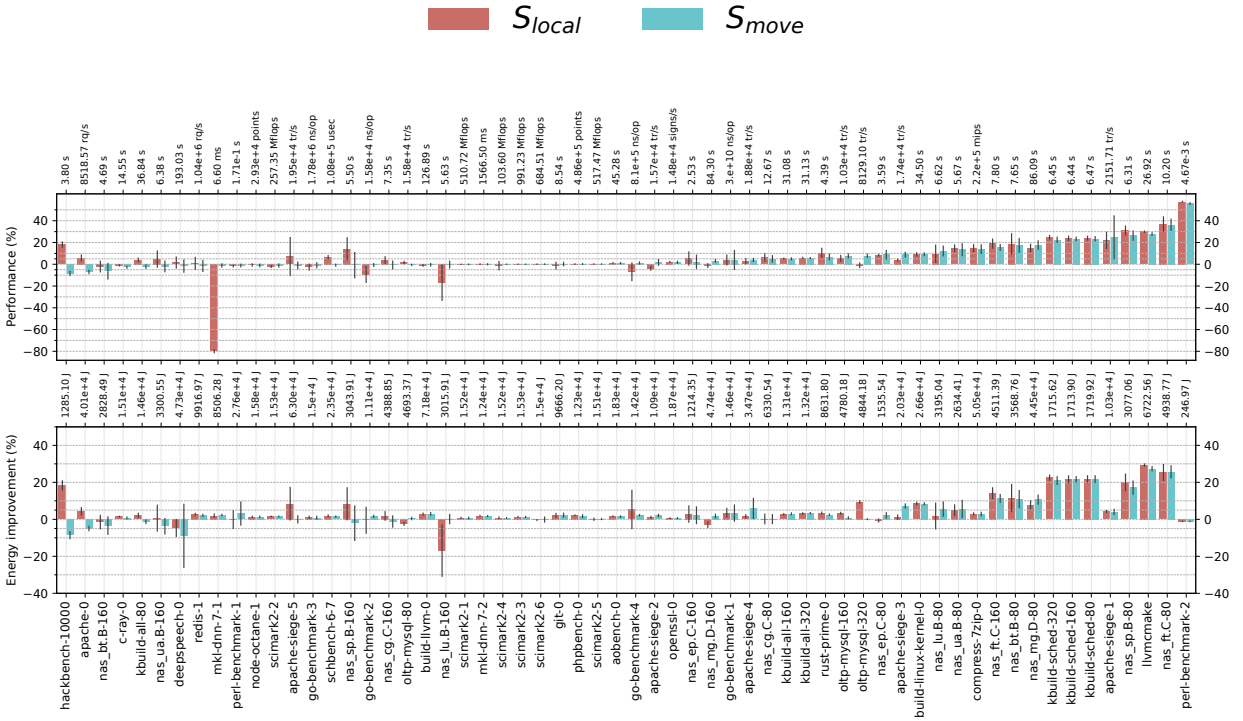
advantage of metrics from the scheduler to select the best frequency. We do not evaluate the `performance` or `ondemand` governors, since the former fixes the frequency while the latter is not supported on our Intel® CPU.

First, we present the complete results on the Intel® server machine and summarize the results on the AMD® desktop machine. We then detail some particularly interesting results, either because our strategies perform well with these benchmarks (`kbuild`), or because they expose the shortcomings of our strategies (`mkl`, `hackbench`). Finally, we discuss the potential overhead of the S_{move} strategy.

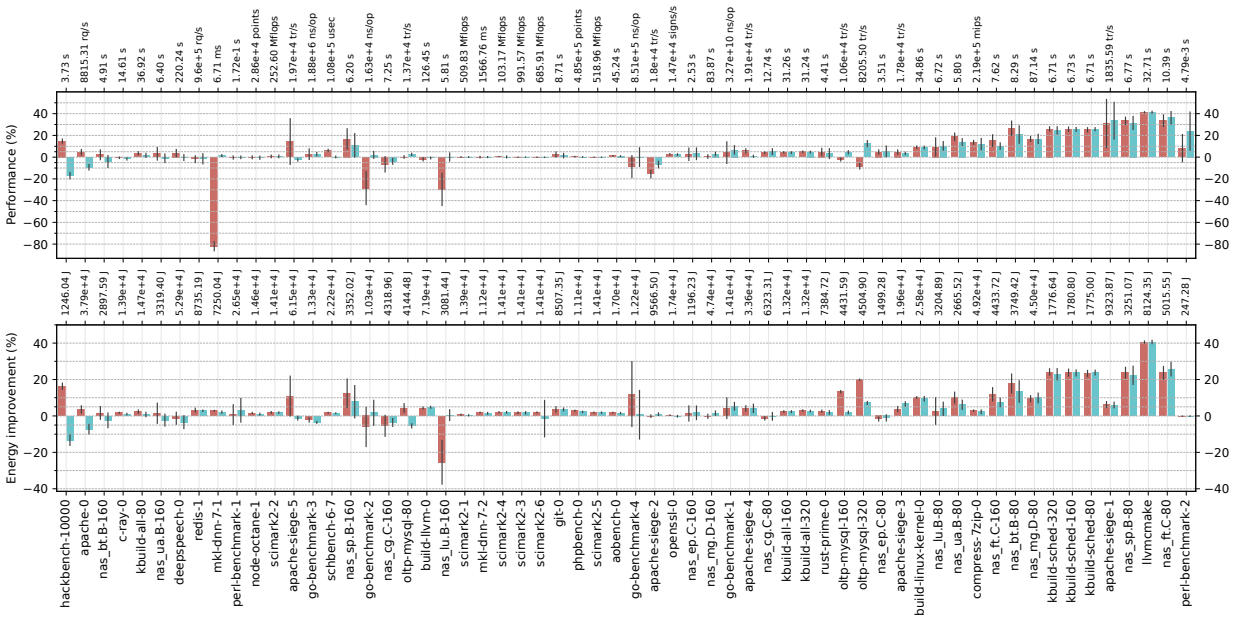
4.6.1 *Execution with the `powersave` Governor*

We first consider the execution under `powersave`. Figure 4.8a shows the performance and energy consumption improvements of S_{local} and S_{move} as compared to CFS on both machines. We consider that improvements or deteriorations that do not exceed 5% to be on par with CFS.

PERFORMANCE. Both S_{local} and S_{move} strategies perform well overall, with respectively 27 and 23 out of 60 applications outperforming CFS. As expected, the best observed results for these policies are seen on benchmarks that extensively use the `fork()/wait()` pattern, and therefore exhibit a large number of frequency inversions. In the best case, S_{local} and S_{move} respectively improve performance by up to 58% and 56% on `perl-benchmark-2`. This benchmark measures the startup time of the `perl` interpreter, and therefore greatly benefits from avoiding frequency inversions since it mostly consists of `fork()/wait()` patterns. In terms of losses, both strategies deteriorate the performance of only three applications, but on very different scales. S_{local} deterio-



(a) Comparison with CFS using the `powersave` governor.



(b) Comparison with CFS using the `schedutil` governor.

Figure 4.8: Performance and energy improvement of S_{local} and S_{move} as compared to CFS, on the Intel® Xeon E7–8870 v4 server.

rates `mkl-dnn-7-1` by 80% and `nas_lu.B-160` by 17%, while S_{move} has a worst-case deterioration of 8.4% on `hackbench`.

ENERGY. Overall, both S_{local} and S_{move} improve energy usage. Out of our 60 applications, we improve energy consumption by more than 5% for 16 and 14 applications, respectively, compared to CFS. Most of the improvements are seen on benchmarks where performance is also improved. In these cases, the energy savings are most likely due to the shorter execution times of the applications. However, we also see some improvements on applications where the performance is on par with that of CFS. This is due to the fact that we avoid waking up cores that are in low power states, therefore saving the energy necessary to power up and run those cores. In terms of loss, S_{local} consumes more energy than CFS on only one application, `nas_lu.B-160`. This loss is explained by the bad performance of S_{local} on this application. This benchmark's metric is its runtime, and increasing the runtime without correspondingly reducing the frequency increases the energy consumption. S_{move} consumes more energy than CFS on two applications: `hackbench`, because of the performance loss, and `deepspeech` that has too high a standard deviation for its results to have significance.

OVERALL SCORE. To compare the overall impact of our strategies, we compute the geometric mean of all runs, where each run is normalized to the mean result of CFS. S_{move} has a performance improvement of 6%, a reduction in energy usage of 3% and an improvement of 4% with both metrics combined. S_{local} has similar overall scores (always 5%), but its worst cases suggest that S_{move} is a better option for a general-purpose scheduler. These small differences are expected because most of the applications we evaluate perform similarly with CFS and with our strategies. We also evaluate the statistical significance of our results with a t-test. With p-values of at most $3 \cdot 10^{-20}$, we deem our results statistically significant.

4.6.2 *Execution with the `schedutil` Governor*

Next, we consider execution under the `schedutil` governor. Before evaluating S_{local} and S_{move} with this governor, we compare it with the `powersave` governor. Figure 4.9 shows the performance and energy improvements of the `schedutil` governor compared to the `powersave` governor with CFS. We omit raw values since they are already available in Figures 4.8a and 4.8b.

Overall, we observe that the `schedutil` governor deteriorates the performance of most applications while improving energy usage. This indicates that this new governor is more aggressive in terms of power savings than the one implemented in hardware by Intel®. This is probably due to this governor having a longer FTL_R and shorter FTL_L

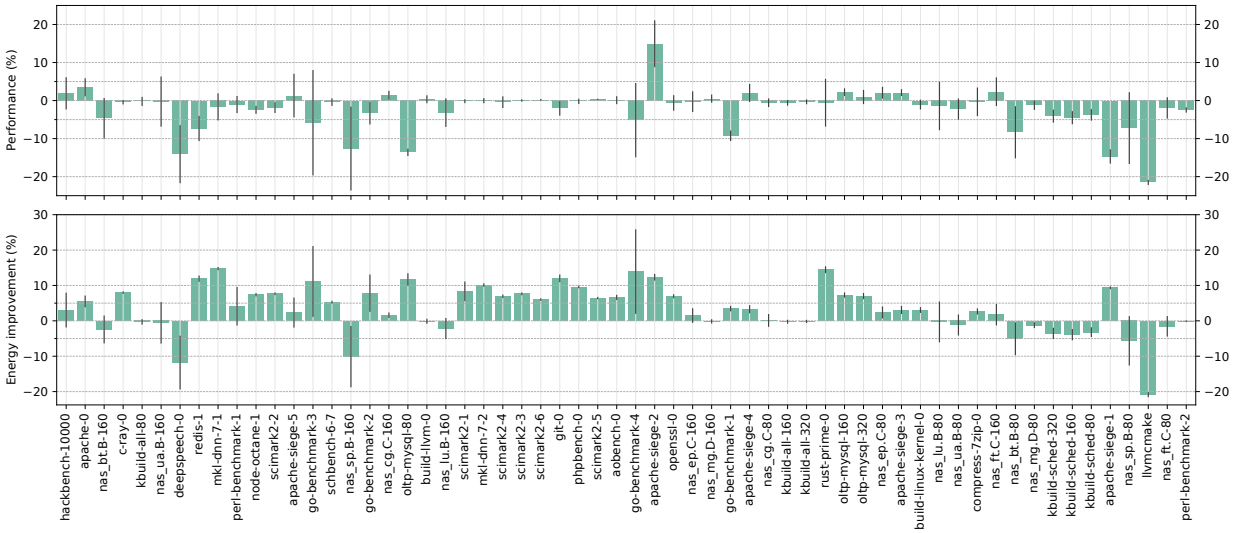


Figure 4.9: Performance of `schedutil` as compared to `powersave` with CFS on the Intel® Xeon E7–8870 v4 server machine.

than the `powersave` governor. This behavior gives us the intuition that frequency inversions may be more frequent with the `schedutil` governor.

Keeping this information in mind, we now evaluate our strategies with this new governor. Figure 4.8b shows the improvement in terms of performance and energy consumption of our strategies compared to CFS when using the `schedutil` governor.

PERFORMANCE. S_{local} and S_{move} outperform CFS on 22 and 20 applications out of 60 respectively. The applications concerned are the same that were improved with the `powersave` governor. In terms of performance losses, however, S_{local} is more impacted by the `schedutil` governor than S_{move}, with 7 applications performing worse than CFS versus only 2. Overall, this is quite similar to what we observe with the `powersave` governor. This means that, even if the `schedutil` governor privileges saving power over performance, it has only a minimal impact on the performance of our strategies. This reinforces the idea that we solve a different problem than scaling governors with our strategies.

ENERGY. The overall improvement in terms of energy usage of `schedutil` with CFS would suggest that we might see the same trend with S_{local} and S_{move}. And indeed, the results are quite similar to what we observe with the `powersave` governor.

OVERALL SCORE. The geometric means with this governor are the following for `schedutil` and S_{move}: 6% for performance, 4% for energy and 5% with both metrics combined. S_{local} has similar results (2%,

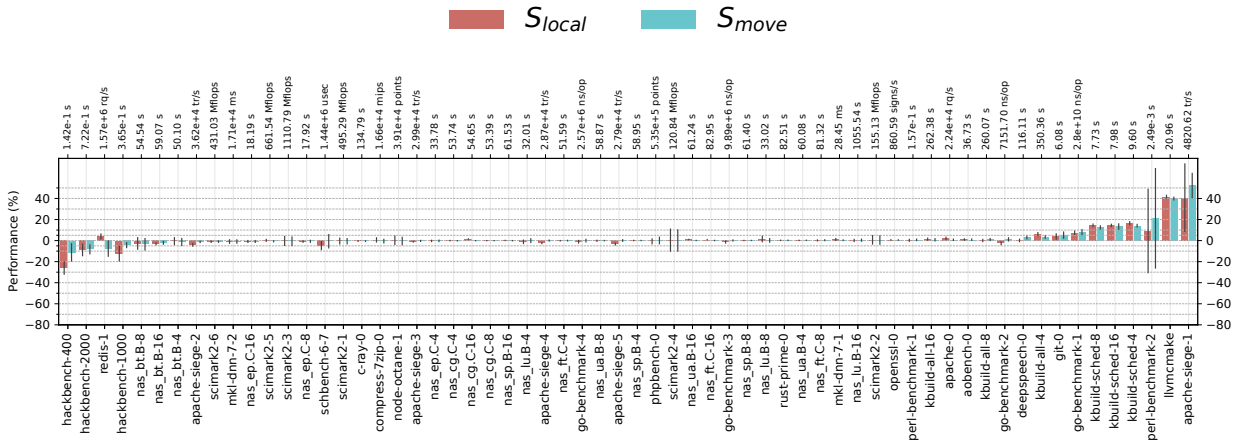


Figure 4.10: Performance improvement of S_{local} and S_{move} compared to CFS on the AMD® Ryzen 5 3400G desktop machine.

6% and 4% respectively), but the worst cases are still too detrimental for a general-purpose scheduler. These results are also statistically significant with p-values of at most $3 \cdot 10^{-20}$. From these results with both governors, we conclude that our strategies to prevent frequency inversions are useful with any governor that dynamically scales the frequency of cores.

4.6.3 *Evaluation on the Desktop Machine*

We also evaluate our strategies on the smaller 8-core AMD® desktop CPU presented in Table 4.2. In contrast to Intel® CPUs, the `powersave` governor on AMD® CPUs always uses the lowest available frequency. Since we need frequency to vary for our strategies to make sense, this governor is not usable in our context. We therefore use the `schedutil` governor on this machine.

PERFORMANCE. As shown in Figure 4.10, we observe the same general trend as on our server machine. S_{local} and S_{move} behave similarly when there is improvement, and S_{move} behaves better on the few benchmarks with a performance degradation. We measure at worst an 11% slowdown and at best a 52% speedup for S_{move}, with an aggregate performance improvement of 2%. Additionally, S_{move} improves the performance of 7 applications by more than 5% while only degrading the performance of 4 applications at the same scale.

The S_{local} strategy gives the same results regarding the number of improved and degraded applications, but suffers worse edge cases. Its best performance improvement is 42% while its worst deterioration is 25%, with an aggregate performance improvement of 1%.

We conclude that even if there is no major global improvement, S_{move} is still a good strategy to eliminate frequency inversions on machines

with smaller core counts. Our performance results are statistically significant, with p-values of $5 \cdot 10^{-4}$ for S_{move} and $3 \cdot 10^{-2}$ for S_{local}.

ENERGY. When we measured the energy consumption on this machine, all results had a large variance on all three strategies (CFS, S_{local} and S_{move}). We did not observe such a variance on our Intel® machine. After some investigation, we suspect that these results are due to the lack of support for the RAPL hardware counters for AMD® CPUs in Linux. Due to this problem, we do not present energy results for this machine.

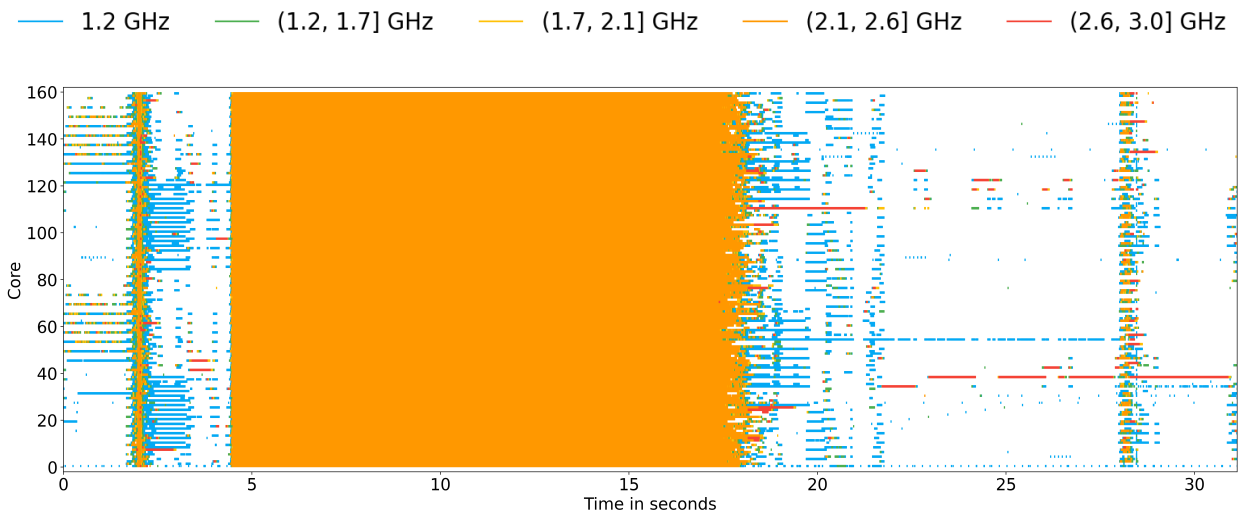
4.6.4 *In-Depth Analysis of Some Benchmarks*

We now present a detailed analysis of specific benchmarks that either performed particularly well or particularly poorly with our solutions. We analyze the trace of the kernel compilation as a favorable benchmark, in echo to our case study in Section 4.1. We then analyze the traces of `mkl` and `hackbench` because they expose specific problems of S_{local} and S_{move} respectively. In this section, we only present traces obtained with the `powersave` governor on the Intel® server machine. Our observations still stand with the `schedutil` governor.

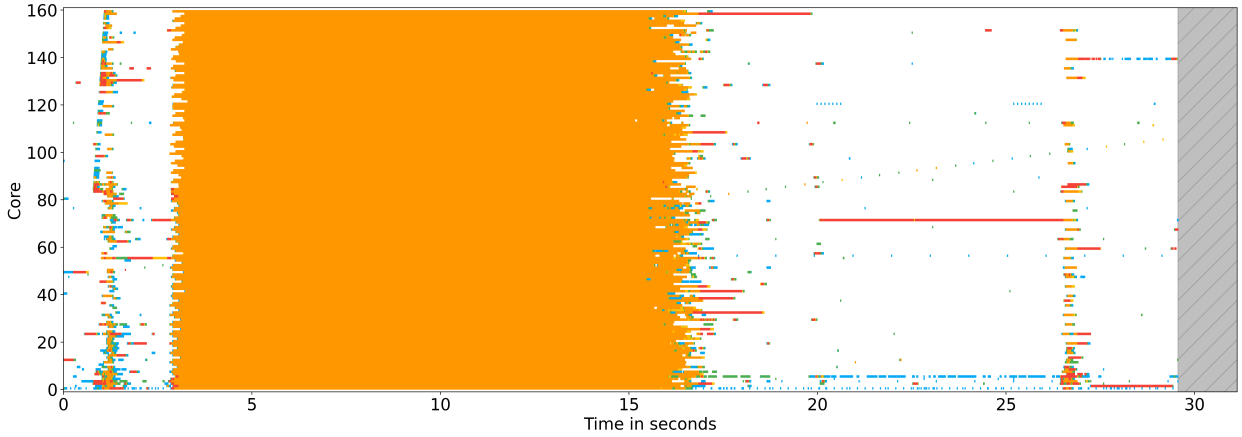
4.6.4.1 *Kernel Compilation*

The kernel compilation benchmark is the same one presented in the case study in Section 4.1. In the previous results, it is referenced as `kbuild-all-320`. Figure 4.11 shows the traces with CFS, S_{local} and S_{move}. During the mostly sequential phases with multiple cores running at a low frequency on CFS (0-2 s, 2.5-4.5 s, 17-27 s), S_{local} and S_{move} use fewer cores at a higher frequency. Both S_{local} and S_{move} behave similarly and avoid frequency inversions due to the `fork()/wait()` pattern. For S_{local}, this is done by always placing new threads locally. For S_{move}, as the waker thread calls `wait()` shortly after the `fork()`, the S_{move} timer does not expire and the woken threads remain on the local core running at a high frequency, thus avoiding frequency inversions. As a result, phase A, as described in Section 4.1, is executed in 4.4 seconds on CFS and in only 2.9 seconds with S_{move}.

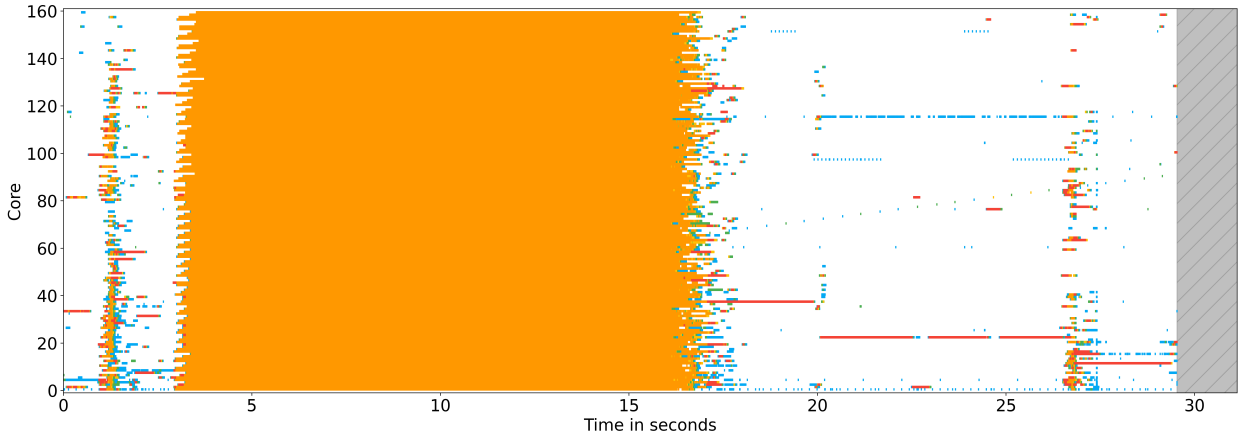
For a clearer a picture of the improvement, Figure 4.12 shows the compilation of the scheduler subsystem of the Linux kernel only (previously referenced as `kbuild-sched-320`). Here, the highly parallel phase is much shorter than with a complete build, as there are fewer files to compile, making the sequential phases of the execution more visible. Again, we see that with S_{local} and S_{move}, fewer cores are used, at a higher frequency. This shows that both strategies solve the frequency inversion problem due to the `fork()/wait()` pattern and behave similarly to the $local_{ipa}$ policy presented in the beginning of this chapter.



(a) With CFS

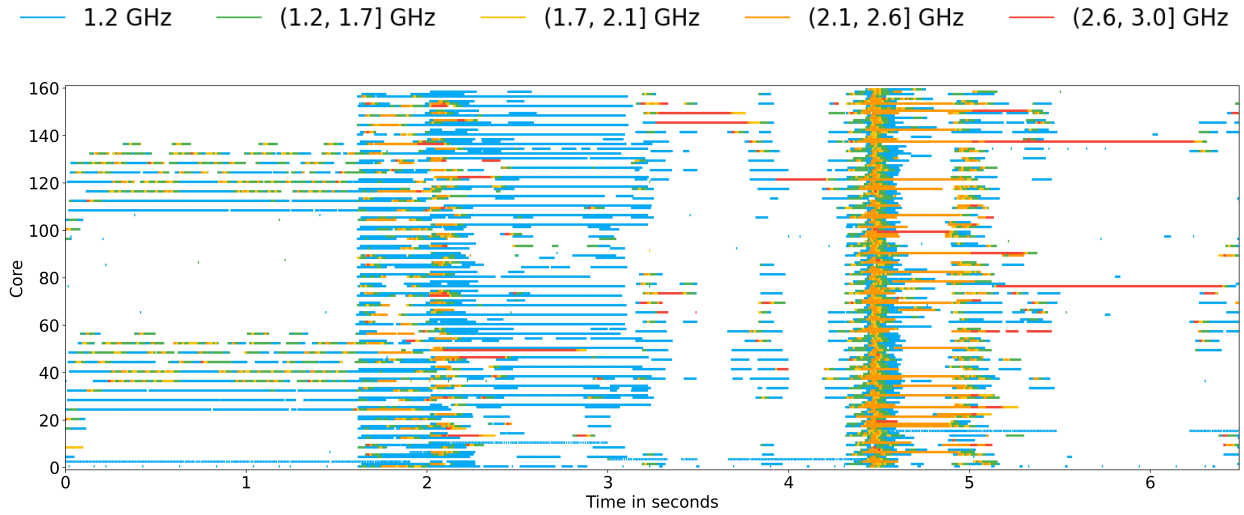


(b) With S_{local}

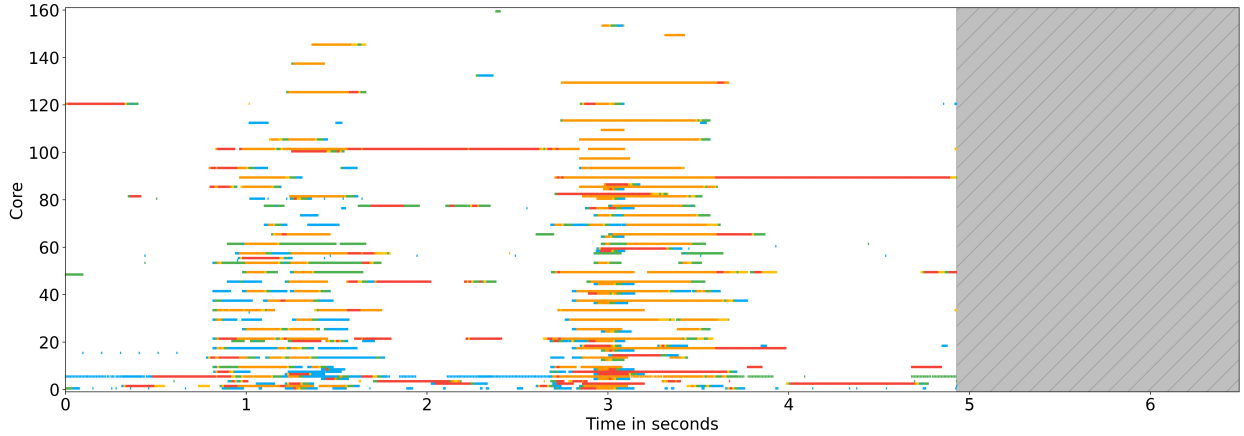


(c) With S_{move}

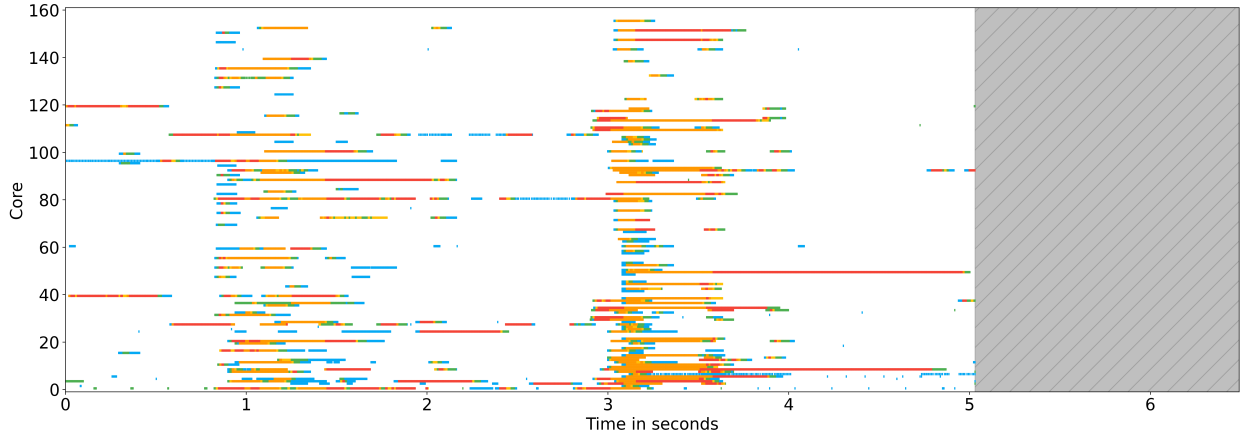
Figure 4.11: Execution trace when building the Linux kernel version 5.4 using 320 jobs.



(a) With CFS

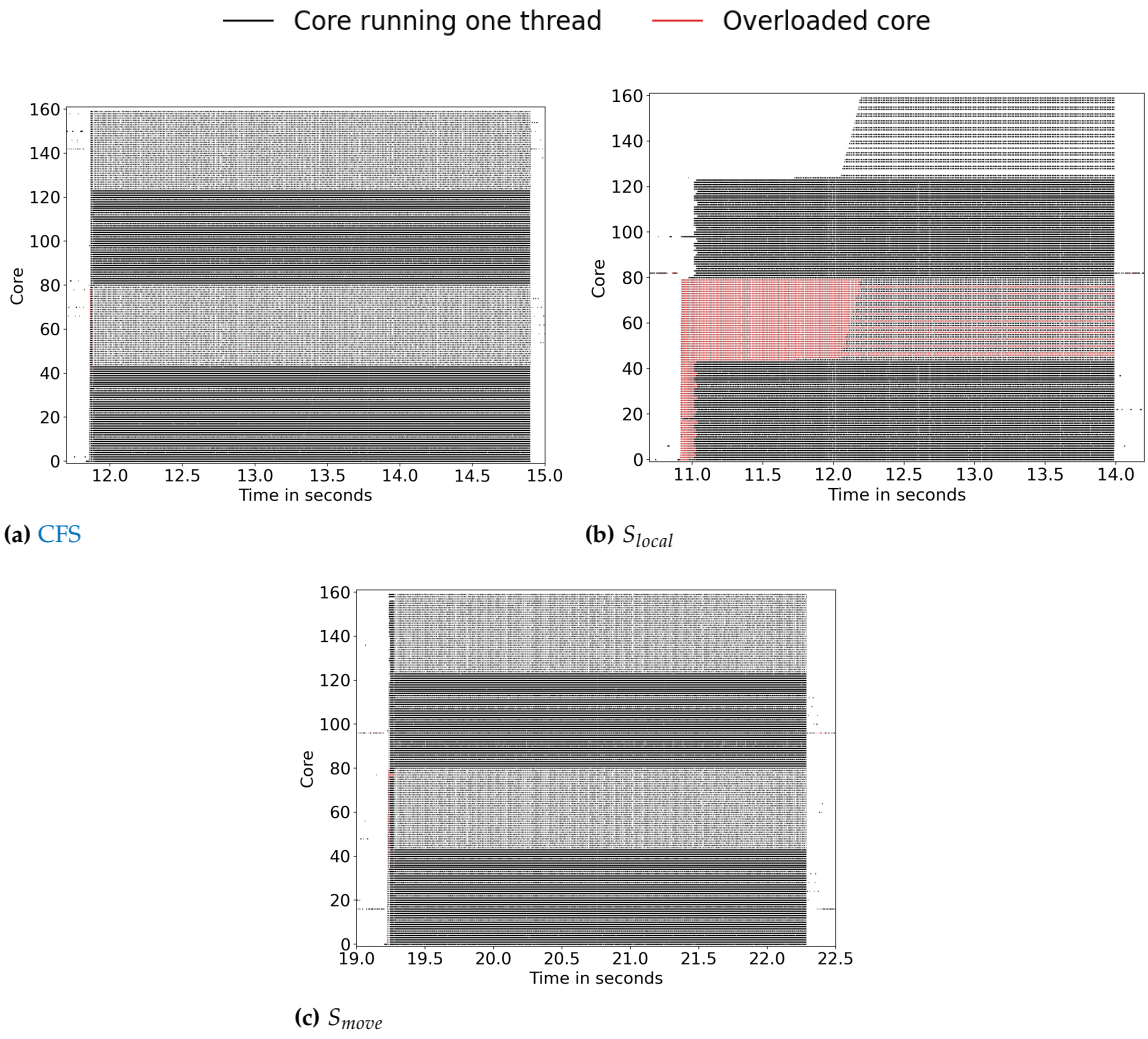


(b) With S_{local}



(c) With S_{move}

Figure 4.12: Execution trace when building the scheduler subsystem of the Linux kernel version 5.4 using 320 jobs.



(a) CFS

(b) S_{local}

(c) S_{move}

Figure 4.13: Number of threads per core during the execution of `mkl-dnn-7-1`.

4.6.4.2 *MKL*

The second application we study in more detail is `mkl-dnn-7-1`, a benchmark of the Intel® Math Kernel Library for Deep Neural Networks. This benchmark is the worst-case scenario for S_{local}: all threads keep blocking and unblocking, therefore avoiding periodic load balancing and keeping running on the same set of cores. Thus, threads that are sharing a core with another thread will tend to remain there with the S_{local} strategy. Figure 4.13 shows the number of threads on the runqueue of each core with all three schedulers with the `powersave` governor. A black line indicates that there is one thread in the runqueue, and a red line indicates that there is more than one, i. e. the core is overloaded. CFS spreads the threads on all cores rapidly, and achieves a balanced machine with one thread per core in less than 0.2 seconds. On the other hand, S_{local} tries to maximize core reuse and

oversubscribes 36 cores. This leads to never using all cores, achieving at most 85% CPU utilization with multiple cores overloaded. This is a persistent violation of the work conservation property, i. e. no core is idle if a core has more than one thread available in its runqueue.

Interestingly, in our experiment, the balancing operations that spread threads are due to system or daemon threads, e. g. `systemd`, that wake up and block immediately, thus triggering an idle balancing from the scheduler. On a machine with nothing running in the background, we could have stayed in an overloaded situation for a long period of time, as ticks are deactivated on idle cores, removing opportunities for periodic balancing. We can see the same pattern on `nas-lu.B-160`, another benchmark that does not work well with S_{local}. S_{move} solves the problem by migrating, after a configurable delay, the threads that overload cores to available idle cores.

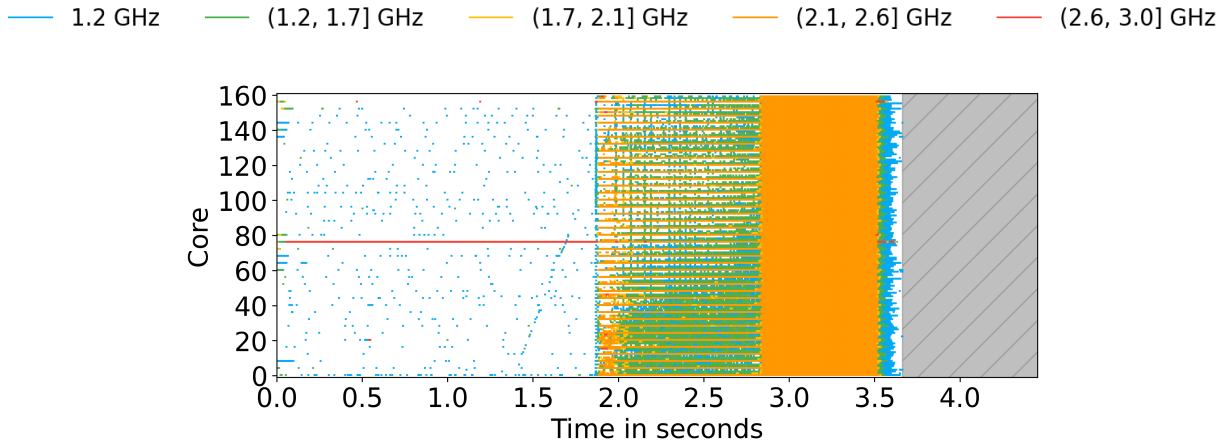
4.6.4.3 *Hackbench*

As a part of the Linux Test Project [108], this microbenchmark is largely used in the Linux kernel scheduler community for testing purposes. It creates groups of threads that communicate through pipes. This behavior makes this benchmark volatile, with a very large number of scheduling events happening.

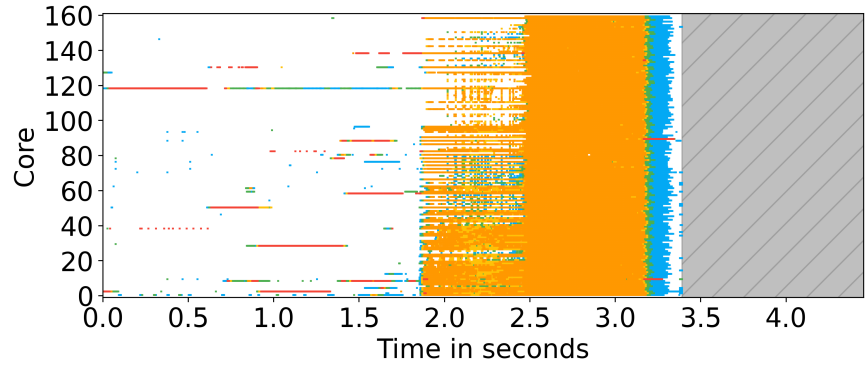
`hackbench-10000` is the worst application performance-wise for the S_{move} strategy. This microbenchmark is particularly stressful for the scheduler, with 10,000 running threads. However, the patterns exhibited are interesting to better understand the shortcomings of S_{move} and give insights on how to improve our strategies.

This benchmark has three phases: thread creation, communication and thread termination. Figure 4.14 shows the frequency of all cores during the execution of `hackbench` with CFS, S_{local} and S_{move}. The first phase corresponds to the first two seconds on all three schedulers. A main thread creates 10,000 threads with the `fork()` system call, and all children thread immediately wait on a barrier. With CFS, child threads are placed on idle cores that become idle again when the threads arrive at the barrier. This means that all cores remain mostly idle. This also leads to the main thread remaining on the same core during this phase. However, S_{local} and S_{move} place the child threads locally, causing oversubscription of the main thread's core and migrations by the load balancer. The main thread itself is thus sometimes migrated from core to core. When all threads are created, the main thread releases the threads waiting on the barrier and waits for their termination, thus beginning the second phase.

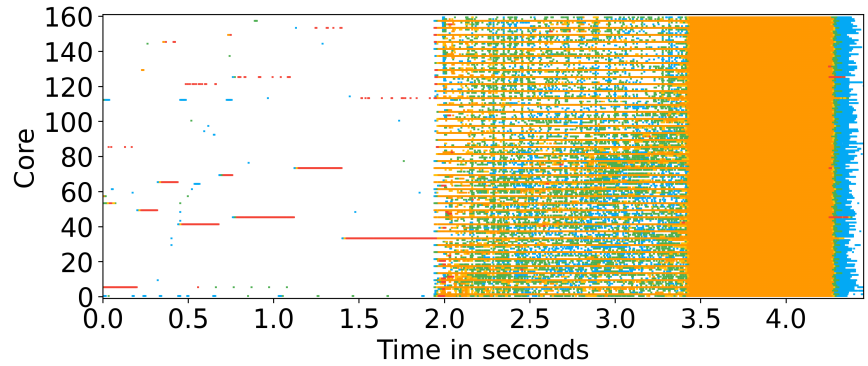
During this phase, the child threads communicate by reading and writing into pipes. CFS tries to even out the load between all cores, but its heuristics give a huge penalty to migrations across NUMA nodes, so a single node runs at a high frequency (cores 0, 4, 8, ... share the same node on our machine) while the others have little work



(a) CFS



(b) S_{local}



(c) S_{move}

Figure 4.14: Execution trace of `hackbench-10000`.

to perform and run at lower frequencies. This phase finishes at 2.8 seconds. The remainder of the execution is the main thread reaping its children and terminating.

S_{local} packs threads aggressively, leading to long runqueues in the second phase, and therefore facilitating load balancing across nodes because of the large induced overload. However, S_{local} still does not

use all cores, mainly avoiding running on SMT pairs of cores (cores n and $n+80$ are SMT siblings on our machine). S_{local} runs the second phase faster than CFS, terminating it at 2.5 seconds, because it uses half of the cores at a high frequency all the time, and many of the other cores run at a medium frequency.

On the other hand, S_{move} performs poorly in the second phase, completing it at 3.4 seconds. The behavior seems very close to that of CFS, with one core out of four running at a high frequency. However, S_{move} results in more idleness or lower frequencies on the other cores. This is due to S_{move} placing threads locally: many threads contend for the local core; some are able to use the resource while others are migrated when the timer interrupt is triggered. The delays cause idleness compared to CFS, and the migrations leave cores idle, lowering their frequency compared to S_{local}. Additionally, when threads are migrated because of timers expiring, they are all placed on the same core, and oversubscribe it. For `hackbench`, choosing the middle ground is the worst strategy. We can also note that load balancing is not able to mitigate this situation because of the high volatility of this workload. This problem was also demonstrated by Lozi et al. [113] on a database application with CFS.

This `hackbench` setup is an extreme situation that is unlikely to happen in real life, with a largely overloaded machine (10,000 threads) and a highly volatile application. This microbenchmark is only interesting to study the behavior of our strategies. Still, overall, S_{move} gives better performance than S_{local}.

4.6.4.4 *Conclusion*

These three applications showed the best-case scenario with the kernel compilation and the worst-case scenario for S_{local} and S_{move} with `mkl-dnn-7.1` and `hackbench`. For these two applications, the latter is an extreme situation that is unlikely to happen while the former shows a weakness of S_{local} that is inherent to this strategy by design. This is one of the reasons that lead us to think that S_{move} is a better strategy overall since it does not have such problems. However, S_{move} requires more configuration with the delay D and the frequency threshold. It also introduces the use of high-resolution timers that could be a new source of overhead. This calls for an evaluation of such an overhead if it exists.

4.6.5 *Scheduling Overhead of S_{move}*

The S_{move} strategy uses mechanisms that can possibly induce an overhead for applications. It is necessary to evaluate the impact our strategies can have because of how they are implemented, and not because of the decisions they make. We identify two possible sources of over-

Threads	Latency		Timers triggered
	vanilla	with timers	
64	78	77	2971
128	86	84	13910
192	119	144	63965
256	2292	3188	93001
512	36544	36544	512
768	60224	60480	959
1024	76416	76928	1290

Table 4.3: Latency of `schbench` (99.5^{th} percentile, in μsec) and number of timers triggered.

head: querying the frequency of all cores and using high-resolution timers.

QUERYING THE FREQUENCY. As explained in Section 4.4.4.2, querying the frequency of a core consists in reading two hardware registers and performing some arithmetic operations, as the current frequency is the division of these two registers times the base frequency of the CPU. Even though this is a very small amount of computation compared to the rest of the scheduler, we minimize it further by querying this information at every tick instead of every time it is needed. In our benchmarks, we notice no difference in performance with or without querying the frequency at every tick.

TIMERS. In S_{move}, a timer is armed every time a thread is created or wakes up. Not all of these timers will actually be triggered, some will be canceled if the thread is scheduled. To evaluate the potential overhead of these timers, we run `schbench` on two versions of Linux: the vanilla v5.4 kernel and a modified version with timers armed under the same condition as with S_{move}. Here, however, the timer handler does not migrate the thread as in S_{move}. We choose `schbench` because it performs the same workload as `hackbench` but provides, as a performance evaluation, the latency of the messages sent through pipes instead of the total completion time. Table 4.3 shows the results of this benchmark.

Overall, the 99.5^{th} percentile of latency is the same for both versions of the kernel, except for 256 threads where timers have a negative impact. We can also observe that the number of timers triggered increases with the number of threads but drops after 256 threads. This behavior is expected: more threads means more wake-ups, but when the machine starts being overloaded, all cores run at high frequencies, and

the timers are less frequently armed. This tipping point arrives around 256 threads because `schbench` threads constantly block, meaning that typically fewer than 160 threads are runnable at a time.

CONCLUSION. Both potential sources of overhead proved to not cause perceptible performance change on our set of benchmarks. This makes us confident in our belief that S_{move} has no side effect due to its implementation. Additionally, there is a potential side effect we did not evaluate regarding the use of timers. There might be an overhead on other subsystems that make use of timers in the kernel, without impacting applications. Unfortunately, we find no good way of correctly evaluating this. Even if there was such a side effect, we think it would not be a problem since the performance of applications would not be hurt.

4.7 DISCUSSION

As previously stated, our proposed solutions S_{local} and S_{move} are purposefully simple. We now discuss other more complex solutions to the frequency inversion problem. We also discuss some related solutions that could have an impact on the frequency inversion problem.

POOL OF HIGH FREQUENCY CORES. A possible solution would be to keep a pool of cores running at a high frequency even though no thread is running on them. This would allow threads to be placed on an idle core running at a high frequency instantaneously. This pool could, however, waste energy and reduce the maximal frequency attainable by busy cores. The former is due to the fact that idle cores are put in sleep states to save energy. The latter is due to how Turbo frequencies work, both on Intel® and AMD® architectures: the maximal Turbo frequency diminishes when the number of active cores increases.

Additionally, sizing this pool could be tricky. The relation between the number of active cores and the maximal Turbo frequency is hardware-dependent and can also be influenced by thermal considerations. The energy/performance gain trade-off is also unclear.

TWEAKING THE PLACEMENT HEURISTIC. Changing more deeply the placement algorithm of CFS to account for the frequency of all cores is another possible solution. This would require the addition of a new frequency heuristic in addition to the existing ones, e. g. cache or NUMA locality. This raises the question of which heuristic should have more impact than others on the decision. For example, the trade-off between using a core running at a higher frequency and using a cache-hot core is not clear, and may vary greatly according to the workload and the hardware architecture.

FREQUENCY MODEL. As previously explained, the frequency of one core can have an impact on the frequency of other cores in the system. The relationship between the frequencies of cores is hardware-specific, and can also be impacted by environmental causes such as the temperature, or software behavior such as the use of AVX-512 instructions. If the scheduler were to take frequency-related decisions, it would need to account for the impact its decisions would have on the frequency of all cores. To do this, an accurate model of the frequency behavior of the CPU would be needed. Unfortunately, such models are not currently available and would probably be difficult to create.

CHILD RUNS FIRST. CFS has a feature that may seem related to our solutions: `sched_child_runs_first`. At thread creation, this feature assigns a lower *vruntime* to the child thread, giving it a higher priority than its parent. If CFS places the child thread on the same core as its parent, the thread will preempt the parent; otherwise, the thread will just run elsewhere. This feature does not affect thread placement and thus cannot address the frequency inversion problem.

Interestingly, using this feature in combination with S_{move} would defeat S_{move}'s purpose by always canceling the timer. The strategy would then resemble S_{local}, except that the child thread would always preempt its parent.

TURBOSCHED. IBM currently develops the TurboSched [39] feature to maximize the usage of Turbo frequencies. The idea is to place small jitter tasks on busy cores instead of waking up idle cores. This way, the maximal Turbo frequency is not reduced because of the growing number of active cores. This is similar to what we do with our S_{local} strategy. However, TurboSched only applies this strategy on threads that have been manually tagged by the user, while we apply our strategies on all threads automatically.

4.8 CONCLUSION

In this chapter, we presented the frequency inversion problem in Linux. This problem occurs on multi-core processors with per-core dynamic frequency scaling. Frequency inversion leads to inefficient usage of high frequency cores and may degrade performance. We propose two strategies, S_{local} and S_{move} to prevent this issue in the Linux v5.4 scheduler, CFS. Both strategies only require small changes to the scheduler code, making it easy to port them to other versions of Linux. The evaluation of our strategies on a diverse set of 60 applications shows that solving the frequency inversion problem: *(i)* significantly improves performance on a large number of applications, *(ii)* does not significantly degrade the performance of applications not disturbed by

frequency inversions. As per-core dynamic frequency scaling becomes a standard feature on latest generation processors, we believe that our work will target most future machines.

We also provide an experimental methodology to evaluate the frequency transition latency of a processor with a given scaling governor. We show that depending on the targeted market of the processor, the behavior of the frequency scaling algorithm differs greatly, and that changing the frequency of a core is all but instantaneous.

FUTURE WORK. In order to best use the frequency of each core, we believe that the scheduler should be fully aware of this information. However, since the decisions of the scheduler can have an impact on the future frequencies of the cores, a model of the frequency behavior of the CPU is necessary. This model would be used to predict the frequency from multiple inputs such as the number of active cores, the FTL, the temperature or the instruction set used. Indeed, all of these characteristics have an impact on the frequency.

There is also room for improvement in the scaling governor and in the duration of hardware reconfiguration. As a matter of fact, if the FTL was instantaneous, the frequency inversion problem would automatically disappear.

On a side note, we could also try to tie this work to our Ipanema DSL presented in Chapter 3. This could be done in the form of a formal proof that a scheduler cannot provoke frequency inversions. This would however require a precise model of the frequency behavior of the CPU.

RELATED PUBLICATIONS. The work described in this chapter has been the subject of two publications in an international workshop and an international conference [28, 66].

5

FEATURE-ORIENTED SCHEDULER ANALYSIS

As exposed in the previous chapters of this thesis, the thread scheduler is a core component of the OS that impacts the performance of applications. The variety of scheduling approaches shows that there exists numerous scheduling needs that require different solutions. These needs are influenced by the workloads and the hardware they are executed on.

In Chapter 3, we proposed Ipanema, a DSL that eases the development of schedulers. With these new tools, we should be able to quickly develop schedulers tailored for a given application. However, being able to easily write a scheduler does not help in writing the good scheduler for an application. This requires knowledge of the behavior of the application and a lot of testing.

In Chapter 4, we proposed tools to help detect and fix performance bugs in Linux, with the example of frequency inversion. Our experimental results showed that our solutions enhanced the performance of most applications. However, some applications showed no improvement, and some even saw their performance worsened. This showed that performance enhancements in the scheduler algorithm are not inherently good for all applications. Some applications require more specific scheduling algorithms to perform well.

Multiple studies available online show that the same application can achieve a different level of performance depending on the OS used, and by extension the scheduler used [60, 100]. Academic researchers also show such results on production OSs [23]. As a software developer, it is not an easy task to choose which scheduler would provide the best performance for a given application. Usually, developers settle for the default scheduler to ensure portability across platforms. But ideally, each application would use the scheduler that obtains the best possible performance.

In this chapter, we set out to produce a methodology with this exact purpose. Creating an **application-specific scheduler** requires knowledge of the behavior of the application as well as of the hardware it will be executed on. It also requires the ability to evaluate which feature of a scheduler is beneficial or detrimental to the application performance.

We first demonstrate the impossibility of doing this feature evaluation with CFS. We propose to do this in a more generic way, not using the code base of CFS. We model schedulers as feature trees with independent features. We then use this model to evaluate features individually and create **application-specific schedulers** tailored for a given application running on a given hardware. We finally devise various methodologies to build the most efficient scheduler for an application from this feature model.

5.1 FEATURE ANALYSIS OF CFS

The scheduler of Linux, CFS, is a general-purpose scheduler that does not target a particular type of application. It is supposed to perform well on all workloads. To do this efficiently, in addition to its core algorithm, CFS implements a large number of smaller features. These features optimize specific code patterns that are not generic to all applications. We presented some of these features in Section 2.5.2.3.

Since CFS is generic, performs well with most applications and already has a large number of features, it seems like a good idea to build our schedulers from its code. To do this, we need to isolate each feature of CFS and evaluate them individually. While doing this, we faced three problems: *(i)* the size of the code base of CFS, *(ii)* the number of actual schedulers after the preprocessing phase, and *(iii)* the overlap between features.

5.1.1 *Size of the Code Base*

The scheduler subsystem of the Linux kernel is a complex piece of software. It has greatly evolved over the years, with multiple algorithms succeeding each other: round robin, $O(n)$, $O(1)$ and CFS. Figure 5.1 shows the evolution of the number of lines of code of CFS since its introduction in Linux v2.6.23 in 2007. We use the `cloc` tool [36] to count the number of lines of code and comments, excluding blank lines. Note that this is only an approximation of the real numbers since we only process files that are exclusively devoted to the scheduler. Scheduler-related code that is present elsewhere is not counted.

At the time of its introduction, CFS was relatively small, with 6,706 lines of code. In version 5.7, the subsystem amounts for 26,213 lines of code.[47] In the span of 14 years between the two versions, the size of the code base of the scheduler was multiplied by 3.9. The evolution is quite steady, with a few hundred new lines per version overall.

[47] This is the last released version as of June 2020.

We can see two exceptions to this steadiness: versions 2.6.25–26 and versions 3.13–14. Both couples of versions introduce more than a thousand lines per version. The former introduces group scheduling that we presented in Section 2.5.2.3, while the latter introduces the deadline scheduling class presented in Section 2.5.3.

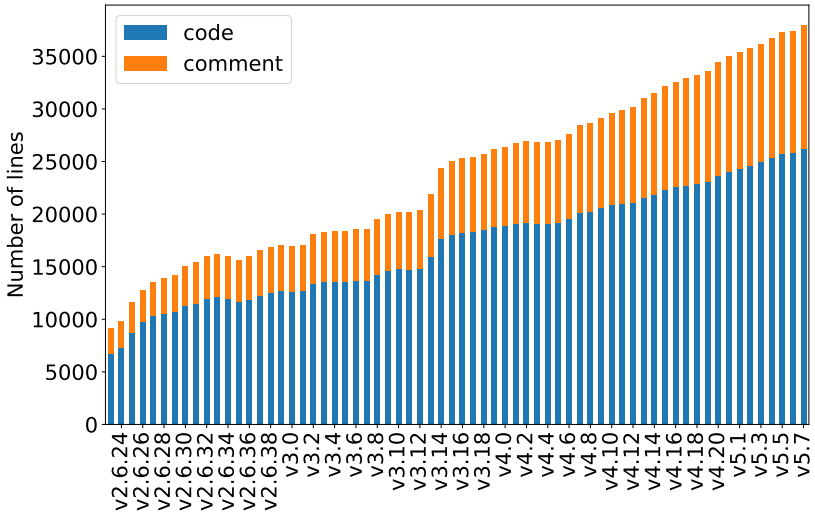


Figure 5.1: Number of lines of the scheduler subsystem of Linux since the introduction of CFS.

The large code base of CFS and its 14 years of steady small additions make it difficult to parse in order to identify every feature manually. The fact that the Linux kernel is a non-trivial piece of low-level C code does not favor this enterprise either.

5.1.2 *Static Configuration*

The Linux kernel supports a large number of hardware with different capabilities. It can also be largely tweaked to change its behavior. This is done with static configuration options in the form of macros that are defined (or not) and the use of `#ifdef` or `#ifndef` blocks. In the scheduling class of CFS alone, i. e. the `kernel/sched/fair.c` file, there are 14 different configuration macros in use as of Linux v5.7. The total number of possible configurations is at most 16,384, assuming there are no dependencies between macros.[48]

All these combinations are actually the number of schedulers we can generate from the code base of CFS, without code modification, only with static configuration options. Depending on the configuration, the resulting code can have different sizes, as shown in Figure 5.2. The number of lines of the `kernel/sched/fair.c` file, after preprocessing the static configuration options, ranges from 2,280 to 9,848 depending on the configuration.

This shows that CFS is polymorphic: it is not one single scheduler, but a multitude of schedulers. Identifying features is furthermore complicated by this since they might be implemented differently depending on the selected configuration options.

[48] We compute the sum of the 0-to-14 combinations for a set of 14 elements.

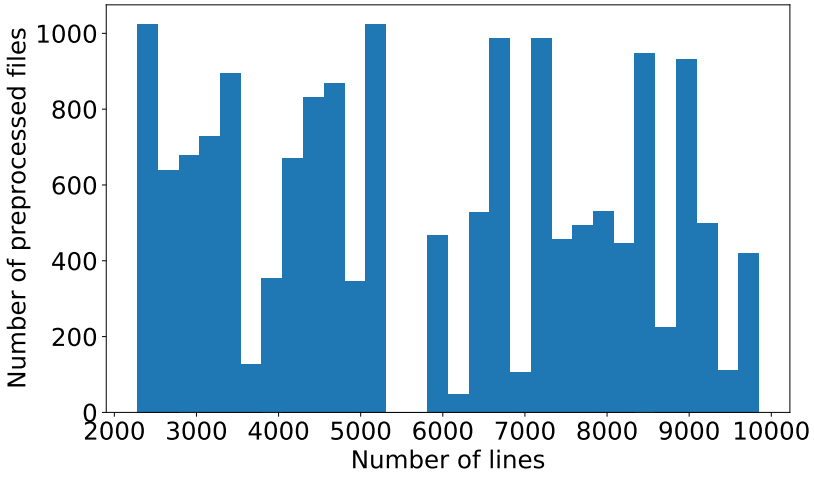


Figure 5.2: Distribution of the size of the `kernel/sched/fair.c` file of Linux v5.7 depending on the configuration options used.

5.1.3 *Overlapping Features*

Finally, even if we are able to identify each feature of CFS, analyzing their performance individually requires to be able to isolate them. This is unfortunately impossible since each feature is not implemented as a separate piece of code. There are multiple occurrences of features that are theoretically distinct, i. e. one could be used without the other and vice versa, but their interwoven code makes it impossible to distinguish one from the other.

To illustrate this, let's only consider the features that can be enabled with a configuration macro. Figure 5.3 shows, for each configuration option, the lines that are enabled only with the option set in green and the lines enabled only with the option unset in red. We can observe that some areas of code are enabled only with a given set of configuration options enabled, creating a form of dependency between features.

We also notice that some features are intertwined with different features across the file. For example, the `CONFIG_FAIR_GROUP_SCHED` option, that enables group scheduling, activates different lines of code when combined with `CONFIG_SCHED_HRTICK`, `CONFIG_CFS_BANDWIDTH` or `CONFIG_SMP`.

All these overlaps and dependencies between features makes them difficult to isolate. Note that the problem gets much harder with features that cannot be enabled or disabled through configuration options. Indeed, we have no way of identifying the exact code that is related to a feature, unless we go through the complete history of commits to find out.

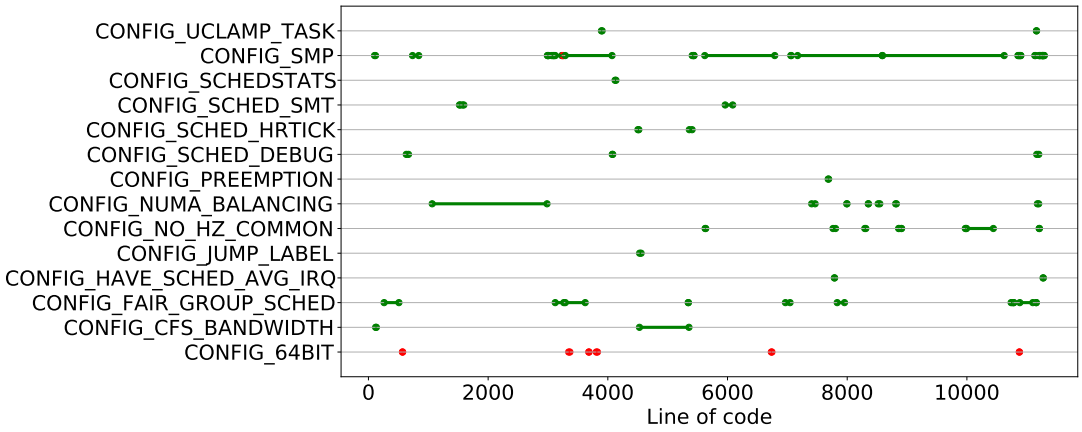


Figure 5.3: Span of influence of each configuration option in the `kernel/sched/fair.c` file of Linux v5.7. Green lines are included if the option is set, red lines are included if the option is not set.

> **Impact of SMPs on the code base**
>
> An interesting observation we can make in Figure 5.3 is that most of the code in this file is related to SMP architectures. Even lines 1500–3500 that do not require the `CONFIG_SMP` are actually related to multi-core systems. Indeed, the `CONFIG_NUMA_BALANCING` option is required for these lines, and this option only makes sense on NUMA architectures, that are necessarily multi-core architectures.
>
> The dependency between these two options is managed by the configuration tool of the kernel, `Kconfig`. Setting the `CONFIG_NUMA_BALANCING` option is impossible if the `CONFIG_SMP` option is not set.

5.1.4 *Conclusion*

Considering the three problems previously described, it seems difficult, if not impossible, to perform a feature analysis of CFS. The quantity and complexity of the code, the large number of configuration options as well as the intertwined features render the feature isolation inconceivable. Additionally, some features can be configured at run time with different values, making the performance analysis even harder.

The impossibility to perform this analysis on CFS motivates us to create a model of a thread scheduler. The main objective of such a model would be to isolate features by design. This would allow us to evaluate and understand the individual impact of each feature on the performance of applications. With this knowledge, we will then

be able to build application-specific schedulers with the best features for a given application.

5.2 FEATURE MODEL OF A SCHEDULER

The formalization of thread schedulers first requires a thorough study of multiple ones. Our study targets the schedulers of general-purpose OSs such as CFS and ULE. We expose multiple recurring features in all studied schedulers. These features are implemented in various ways in order to achieve different performance goals.

We first describe the type of model we use, the feature model. We then describe the model we develop to describe schedulers.

5.2.1 *Feature Model*

We choose the **feature model** described by Kang *et al.* [91] to represent as a formalism the recurring features in the general-purpose OSs we study. This model describes a system as a set of features and sub-features that can be mandatory, optional or alternatives.

Figure 5.4 shows an example of such a model in the form of a feature tree. A feature tree is a representation of a feature model. Each node represents a feature and each level of the tree increases the level of detail of the features. There are three possible relationships between a node and its children:

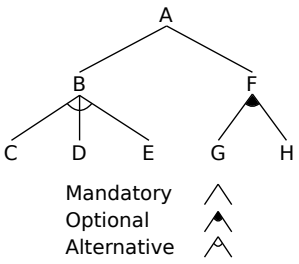


Figure 5.4: Example of feature tree with mandatory, optional and alternative features.

- **mandatory**: all children must be implemented to implement the parent feature. In our example, implementing A requires the implementation B and F.
- **optional**: multiple children can be implemented to implement the parent feature (`or`). Implementing F requires the implementation of G or H, or both, or none.
- **alternative**: only a single child must be implemented to implement the parent feature (`xor`). Implementing B requires the implementation of a single feature among C, D and E.

With this type of model, we can easily express sets of features with relationships between them.

5.2.2 *The Scheduler Feature Model*

From our experience and the previous scheduler study in Chapter 2, we build a model that represents a thread scheduler in the form of a set of features. Figure 5.5 shows the resulting feature model in the form of a feature tree. We also annotate the feature model with two notations: *fixed* and *variable* features. Fixed features (solid

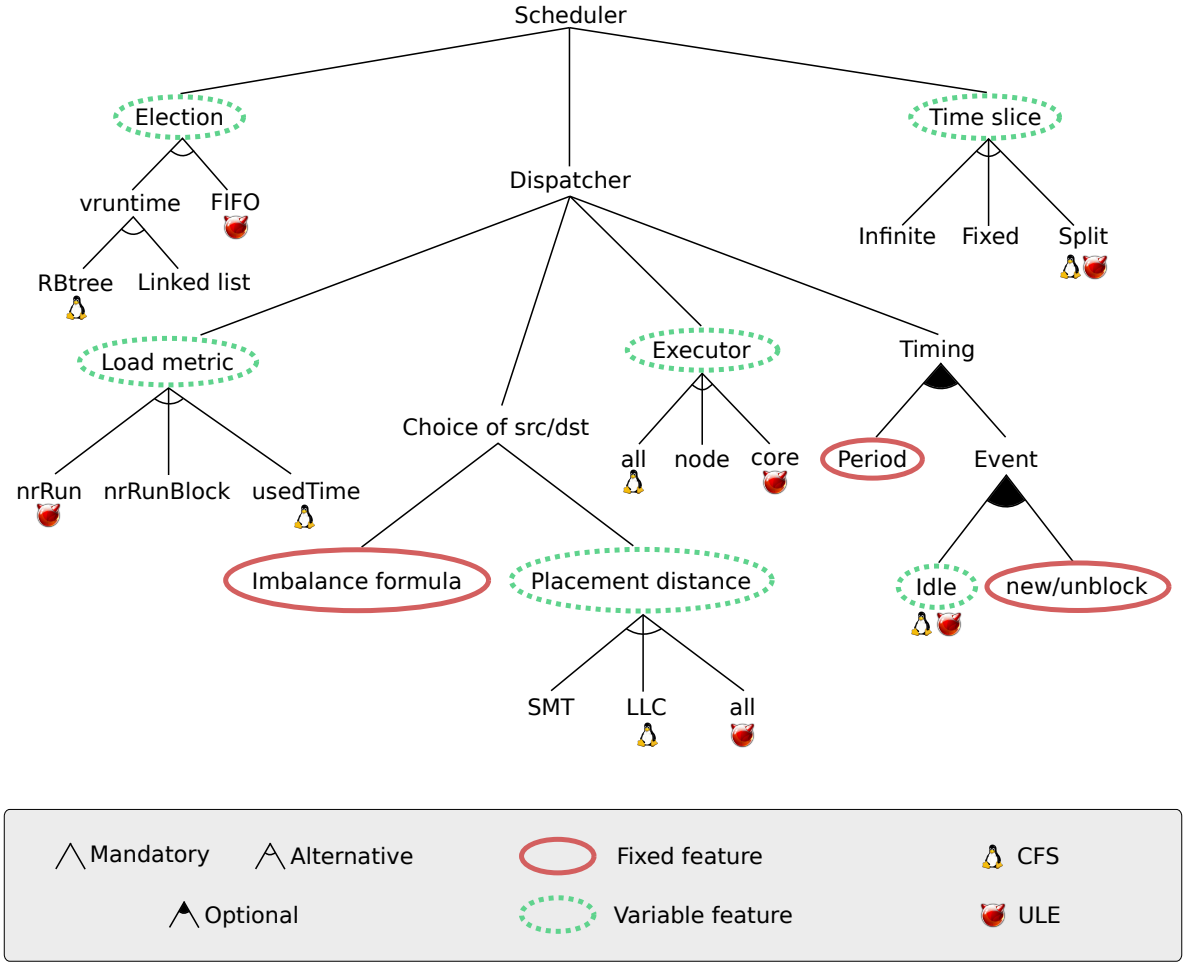


Figure 5.5: Feature tree of a thread scheduler as defined in this chapter.

red circle) are always enabled with a single implementation either because not having them does not make sense in our setup or because they use discrete configuration values that we fix to minimize the experimentation time. Variable features (dashed green circle), on the other hand, have multiple possible implementations that vary in the schedulers we will generate.

Our study of existing thread schedulers exposed two main features: the choice of which thread runs on a core (*election*) and for how long (*time slice*). In addition to these features, we must take into consideration the OS's design at the scheduler level. In our case, we use the Linux kernel to implement our schedulers. On multi-core systems, Linux instantiates one runqueue per core, thus raising the need for a mechanism that places threads on cores. The thread placement policy is determined by the *dispatcher*. This feature has four sub-features: *load metric*, *timing*, *executor* and *choice of source and destination*.

The features that are the closest to what CFS or ULE implement are marked with the corresponding OS logo. The rest of this section describes these features in detail.

5.2.2.1 *Election.*

This feature determines how the scheduler chooses the next thread to run on a core. On each core, threads are stored in a runqueue. We study two possible election mechanisms: choosing the thread that has run the least in the runqueue (*vruntime*) or the thread that arrived first in the runqueue.

The first mechanism is implemented by sorting threads by ascending runtime in the runqueue, i. e. the thread with the lowest execution time has the highest priority. This allows the scheduler to be fair to interactive threads that need to be scheduled frequently for a short period of time. We implement two versions of this mechanism: one that uses a red-black tree [73] (called *rbtree*) and one that uses a circular doubly linked list (called *linked list*). In both versions, threads are sorted by runtime. Red-black trees provide insertion, deletion and pop operations with an $O(\log n)$ average complexity, but may be costly because of the rotations needed to keep the tree balanced. Sorted doubly linked lists provide deletion and pop operations in constant time, but insertion is more costly ($O(n)$) because the list needs to be traversed to place the thread in the correct location and keep the list sorted. CFS implements this mechanism with a red-black tree.

The second mechanism is implemented using a FIFO (First-In First-Out) circular doubly linked list. A FIFO provides insertion, deletion and pop operations in constant time. Since threads are not sorted by runtime but follow a first-in first-out pattern, latency sensitive applications may be delayed if there are a lot of threads in the runqueue. ULE implements a variant of this feature with two FIFO lists, one for interactive threads and one for batch threads.

5.2.2.2 *Time slice.*

When a thread is given the right to run on a core, it owns this right for a duration called a time slice. The *time slice* feature determines how this duration is computed. We consider three alternatives: *infinite*, *fixed* and *split*.

The *infinite* time slice alternative disables preemption. Context switches are driven by the thread voluntarily yielding the CPU or being blocked waiting for a resource. This is present in cooperative scheduling policies.

The *fixed* alternative allocates the same time slice for all threads. We fix the time slice to 10 ms for our experiments.

The *split* alternative allocates a time slice that depends on the number of threads present in the runqueue with the following formula:

$$timeSlice = \begin{cases} X\ ms & \text{if } |threads| > T \\ \dfrac{Y}{|threads|}\ ms & \text{otherwise} \end{cases}$$

CFS and ULE both implement this feature with different values for X, Y and T (on our testing systems, $X = 3$, $Y = 24$ and $T = 8$ for CFS; $X = 4$, $Y = 25$ and $T = 6$ for ULE). Our implementation of this feature follows the definition used by CFS. If the core is overloaded, i. e. more than 8 threads in the runqueue, each thread is allocated a 3 ms time slice. Otherwise, each thread is allocated an equal share of a 24 ms period.

5.2.2.3 *Dispatcher*

The *dispatcher* feature determines the thread placement policy of the scheduler. This includes the metric used to compare runqueues, where to place threads, when threads are migrated and by whom.

LOAD METRIC. This feature is used to measure the amount of work available on a core. This metric allows the dispatcher to compare cores and decide if threads should be migrated from one core to another. We consider three alternative metrics: *nrRun*, *nrRunBlock* and *usedTime*.

nrRun measures the load of a core as the number of threads in the runqueue, i. e. runnable threads, as implemented in ULE.

nrRunBlock takes into account the number of runnable threads as well as the number of threads that blocked last on this core. This allows the scheduler to keep track of threads that blocked on a core and will eventually wake up on it. This can be useful when blocked threads wake up immediately since it prevents the scheduler from migrating a distant thread to balance a load that would have been immediately balanced anyway. However, threads that stay blocked for a long time will weigh on the load and may prevent balancing even if there is no runnable thread in the runqueue.

usedTime defines the load of a thread as the proportion of time the thread spent runnable (*runnableTime*) regarding its allocated time slice (*timeSlice*). This alternative also takes previous loads into account to smooth the load over time: 80% of the load corresponds to the previous load while 20% depends on the values *runnableTime* and *timeSlice* at computation time. This is a simplified version of the decaying load average of CFS. Therefore, the load of a thread at time t ($load_t$) is computed as follows:

$$load_t = 0.8\ load_{t-1} + 0.2\ \frac{runnableTime}{timeSlice}$$

TIMING. This feature is used in the *dispatcher* to choose at which moment threads are migrated among cores. We divide this feature into two sub-features: *period* and *event*.

The *period* feature determines if a load balancing operation should be triggered periodically and, if so, the period between those operations. Since the range of values for the period is quite large (from 0 to $2^{64} - 1$ nanoseconds on a 64-bit machine) and not all values are meaningful

(triggering a load balancing operation every nanosecond might be a little excessive), we choose to fix this feature: all generated schedulers perform a periodic load balancing every n milliseconds, where n is the number of cores on the machine, following the behavior of CFS.[49]

[49] In reality, CFS balances cores hierarchically, and the value of n is the number of cores in the subset of cores being balanced.

The *event* feature triggers migrations when a core has no more threads to run (*idle* event) or when a thread is created or wakes up from a blocking operation like a synchronous I/O (*new/unblock* event). These two features are optional, which means that they can both be enabled at the same time, or both disabled.

The *idle* feature determines if idle balancing is enabled or disabled. When an idle event occurs, i. e. a core has no more thread to execute, the scheduler can either reduce the power consumption of this core by letting it enter a lower power state or perform idle balancing to try to keep it busy. In the latter case, a load balancing operation is triggered on this core to allow it to find pending work on another core.

As for the *new/unblock* event, the scheduler can migrate the thread concerned at this moment. In our schedulers, this event is always enabled because we target multi-core systems. The choice of the destination core of the thread is determined by the *placement distance* feature presented later on.

EXECUTOR. The periodic load balancing algorithm distributes work among cores. In order to do that, load balancing events are triggered periodically on the cores of the machine. The executor alternatives determine the cores that perform load balancing operations.

The *all* alternative allows each core to perform periodic load balancing operations to balance itself with another core of its own choosing. All cores can do this in parallel but may take conflicting migration decisions and fail to balance correctly.

The *core* alternative allows only one core to perform load balancing for all the cores of the machine. Therefore, no conflicting decisions are taken, at the expanse of serializing the whole balancing process and increasing the number of remote memory accesses on NUMA machines.

The *node* alternative allows only a single core per NUMA node to perform load balancing operations. This core performs a load balancing operation for each core in the node. On an n-node machine, n load balancing operations can take place in parallel, thus minimizing the probability of conflicts. This alternative is a good compromise between *all* and *core* alternatives.

CHOICE OF SOURCE/DESTINATION. This feature determines how the source and destination cores of a migration are determined. For migrations due to the load balancing algorithm (either periodic or after an idle event), the destination is the core for which the load balancing is executed. The source, however, is determined by the

imbalance formula sub-feature that defines if two cores need to be balanced. We currently fix this feature and consider that if at least one migration between the source core and a potential destination core can reduce the imbalance between these cores, then they are unbalanced.[50] All our schedulers choose the most loaded core among those unbalanced cores as the source.

For migrations due to a new or an unblock event, the source is the current position of the thread (or of the thread's parent for new threads), and the destination is determined by the *placement distance* sub-feature. In most cases, it should be beneficial to keep a thread close to its most recently used core because the data it was using may still be available in this core's hardware caches. However, this can lead to a load imbalance between cores if threads are always kept on the same subset of cores. The placement distance feature determines how far away from its previous location a thread can be migrated. We implement three alternatives: a thread can only be placed on an SMT sibling (*SMT*), a core in its last level cache domain (*LLC*), or any core on the entire machine (*all*). The core selected is the least loaded core among the cores that respect the placement distance.

[50] The imbalance considered here regards the load metric defined in Section 5.2.2.3.

5.3 FEATURE EVALUATION

Now that we have defined an extendable feature model able to capture a large number of features found in general-purpose schedulers, we evaluate the impact of each of these features on the performance of applications. To do so, we implement each feature described in our model independently and generate all possible combinations of features. With the current state of our model, we are able to generate 486 schedulers by combining the 16 features that have variable implementations (*idle* and the sub-features of *election*, *time slice*, *load metric*, *executor* and *placement distance*). The generated schedulers use the `SaaKM` API presented in Section 3.4.

With this new collection of schedulers available, we can evaluate the performance of individual features on given applications. Additionally, we perform a stability analysis that will help us detect variability in the performance of applications due to the scheduler.

5.3.1 *Experimental Setup*

For our experiments, we use a set of 20 applications from five benchmark suites that we select based on two criteria: *application behavior* and *execution time*. The former criterion helps us consider a large number of different behaviors and make sure we cover most existing workloads. The latter stems from a practical problem: the large number of schedulers we evaluate and the small number of machines available

to us makes it impossible to evaluate applications with long execution times. We evaluate on the following applications:

- 7 applications from PARSEC [17]: `blackscholes`, `bodytrack`, `canneal`, `facesim`, `ferret`, `fluidanimate` and `streamcluster`
- 7 applications from the Phoronix benchmark suite [119]: `build-linux-kernel`, `compress-7zip`, `c-ray`, `john-the-ripper-md5`, `openssl`, `stockfish` and `x264`
- `hackbench` from the Linux Test Project [108]
- `oltp` (MySQL), `mutex` and `memory` from `sysbench` [98]
- `bayes` and `nutchindexing` from the HiBench benchmark suite [84] using Apache Hadoop [10]

Table 5.1 presents the behavior of these applications and the performance results we use as a baseline. We split the data into three categories: threads, behavior and performance.

THREADS. A first characteristic is the number of threads used by each application. The *max nr* column reports the maximum number of threads for the benchmark while the *creation rate* column reports when and how threads are created. These two characteristics are important because they have a major impact on the occupation of the CPU resources and on the pressure on the scheduler.

The maximum number of threads is statically selected for each benchmark. For the PARSEC applications and `oltp`, we use the setup with the best performance under CFS. For the HiBench applications, we use the `tiny` inputs (adapted for a single machine Hadoop cluster) with an equal number of map and reduce threads, with the total number of threads matching the number of hardware threads on our machines. For the Phoronix applications, we use the default settings of the benchmark suite (usually the number of hardware threads on the machine or a multiple of that number).

BEHAVIOR. We profile certain metrics when running our applications with CFS to determine their "default" behavior and check that we evaluate our approach on different types of applications. We periodically (every second) sample multiple metrics: CPU usage (system and user), the number of blocking events per second (interactivity), the number of context switches or the number of migrations. Due to the lack of space, we only report two metrics in the table. The 20 applications we select were chosen due to the diversity of their behavior regarding these metrics.

	Threads		Behavior		Performance		
Application	**Max nr**	**Creation rate**	**CPU usage**	**Blocking events**	**Metric**	**CFS**	**CFS + Pin**
bayes	24	3 peaks at 80	low (10%)	medium (1,500/s)	time (s) ↘	$116.2 \pm 3.36\%$	$634.9 \pm 0.64\%$
blackscholes	24	at startup	half low (10%) half high (100%)	low (100/s)	time (s) ↘	$39.8 \pm 0.41\%$	$39.7 \pm 0.54\%$
bodytrack	16	at startup	medium (45%)	medium (8,000/s)	time (s) ↘	$43.1 \pm 1.75\%$	$43.3 \pm 2.59\%$
buildkernel	24	400/s	half low (5%) half high (100%)	medium (5,00/s)	time (s) ↘	$19.8 \pm 4.37\%$	✗
canneal	12	at startup	half at 5%, half at 50%	medium (3,000/s)	time (s) ↘	$60.6 \pm 0.53\%$	$59.6 \pm 0.53\%$
compress7zip	24	4 peaks at 35	4 high peaks (80–100%) (half the time)	4 medium peaks (9,000/s)	Mips ↗	$33.5k \pm 0.77\%$	$22.7k \pm 3.57\%$
cray	384	at startup	high (100%)	low (75/s)	time (s) ↘	$115.3 \pm 0.40\%$	$122.0 \pm 0.06\%$
facesim	16	at startup	low to medium (20–40%)	medium (6,000/s)	time (s) ↘	$168.9 \pm 0.71\%$	$169.9 \pm 0.80\%$
ferret	12	at startup	medium (70%)	medium (1,000/s)	time (s) ↘	$42.2 \pm 0.94\%$	$42.9 \pm 1.01\%$
fluidanimate	16	at startup	medium (50%)	medium (1,000/s)	time (s) ↘	$89.2 \pm 2.43\%$	$83.4 \pm 1.40\%$
hackbench	1000	at startup	high (80%)	high (100,000/s)	time (s) ↘	$0.6 \pm 6.43\%$	✗
johntherippermd5	24	at startup	high (100%)	low (100/s)	checks/s ↗	$303.5k \pm 0.83\%$	$307.2k \pm 0.59\%$
memory	256	at startup	high (100%)	low (200/s)	time (s) ↘	$9.9 \pm 0.52\%$	$9.9 \pm 0.73\%$
mutex	256	at startup	high (100%)	low (300/s)	time (s) ↘	$8.8 \pm 0.74\%$	$8.93 \pm 0.84\%$
nutchindexing	24	50–120/s	low (10%)	medium (1,000/s)	time (s) ↘	$3.3 \pm 9.12\%$	$3.4 \pm 3.57\%$
oltp (latency 95^{th})	64	at startup	medium (70%)	high (150,000/s)	time (s) ↘	$56.6 \pm 2.08\%$	$58.7 \pm 3.55\%$
oltp (avg latency)	64	at startup	medium (70%)	high (150,000/s)	time (s) ↘	$28.0 \pm 3.59\%$	$28.8 \pm 2.37\%$
oltp (throughput)	64	at startup	medium (70%)	high (150,000/s)	tr/s ↗	$2,290.5 \pm 3.57\%$	$2,218.0 \pm 2.40\%$
openssl	24	at startup	high (100%)	low (65/s)	sign/s ↗	$734.8 \pm 0.31\%$	$745.6 \pm 0.57\%$
stockfish	24	25/s	high (100%)	low (100/s)	nodes/s ↗	$19.4M \pm 1.12\%$	$19.4M \pm 1.66\%$
streamcluster	12	10/s	high (100%)	medium (5,000/s)	time (s) ↘	$108.8 \pm 1.73\%$	$96.8 \pm 3.20\%$
x264	24	at startup	low (10%)	medium (2,500/s)	fps ↗	$32.1 \pm 5.53\%$	$29.26 \pm 2.55\%$

Table 5.1: Application behavior and performance with CFS and threads pinned. (↗ higher is better ↘ lower is better ✗ timeout)

PERFORMANCE. We also report the performance of each application. For most applications, the performance metric is the execution time in seconds. However, some applications have a different performance metric, such as the frame rate for `x264` or the number of MD5 checks per second for `johntherippermd5`. Also, `oltp` appears three times because it provides three performance metrics (average latency, 95^{th} percentile latency and throughput).

We present the performance metrics when run with CFS as well as when run with CFS with each thread pinned to a core in a round-robin fashion. The former is for comparison purpose only. The latter, however, shows the performance of CFS when disabling the *dispatcher* feature. The applications with no result (marked ✗ in the table) took too long to run and reached a timeout (more than 100 times the duration of the run with CFS). We observe that only `streamcluster` gains more than 5% of performance when pinning threads to cores, whereas four applications (`bayes`, `compress`, `fluidanimate` and `x264`) lose more than 5% in terms of performance. This means that for 13 applications out of 20 (we count `oltp` once), the placement strategy of CFS has no impact on performance.

SETUP. We run our experiments on a cluster of eight identical machines equipped with a 2-socket Intel® Xeon E5645 (12 physical cores, 24 cores with SMT enabled) and 64 GiB of RAM. All machines run a Debian 8 OS, our customized Linux v4.19 kernel and all our benchmarks and dependencies installed. Each application is always run on the same machine to avoid discrepancies due to hardware differences. For each scheduler (CFS and pinned CFS included), each application is run 10 times.

SCHEDULER GENERATION. We generate our 486 schedulers from the feature model presented in Section 5.2.2. In this model, we have 16 variable feature implementations, i. e. the leaves of the feature tree. In Section 5.1, we showed that the feature analysis of CFS was made impossible by multiple factors, most notably by the overlapping code of multiple features. When implementing the features from our model, we pay particular attention not to have this flaw. Each feature is implemented independently and is compatible with all other features of the model, except among alternatives.

With this many schedulers, a possible experimental problem would be the time needed to run all experiments. Each application is executed 10 times with every generated scheduler, as well as CFS and CFS with pinned threads. Even with a cluster of eight machines, this takes a very long time to run. Changing the scheduler between executions also becomes a real problem if they are implemented with the scheduling class internal API of Linux. We remove this problem by using the `SaaKM` internal API we developed in Chapter 3. With `SaaKM`-enabled

schedulers, we can add and remove scheduling policies at run time, with no need to reboot the machine or compile hundreds of kernels.

5.3.2 *Methodology Overview*

After running each application with all 486 generated schedulers as well as CFS 10 times, we can use the results gathered to analyze the impact of each feature on performance independently. This information is necessary to understand which features improve the performance of a given application and, ultimately, build application-specific schedulers. Before presenting our methodology in detail, we outline the general idea behind the two main phases of this methodology: **stability analysis** and **feature impact analysis**.

STABILITY ANALYSIS. When drawing conclusions from data gathered experimentally, one must first assess the statistical validity of said data. Indeed, if the performance of a given scheduler is not consistent among its 10 runs, it is incorrect to infer a meaningful performance measure from this data. This leads us to design a stability analysis of our schedulers. The primary goal of this analysis is to exclude schedulers that have results with a low statistical significance for further analysis. A secondary goal is to assess the stability of applications. This is made possible by the fact that we have a large number of schedulers to test with.

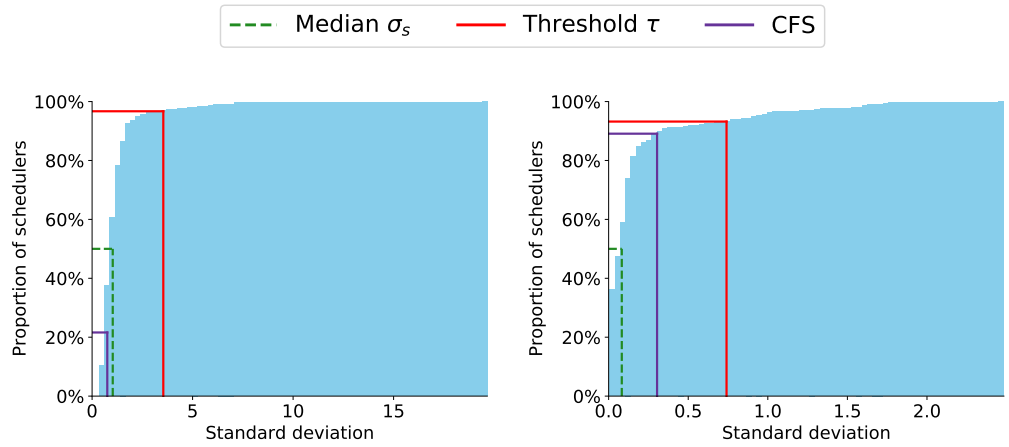
FEATURE IMPACT ANALYSIS. With unstable data out of the way, we can now study the impact of each feature on performance independently. The general idea is to isolate a set of features that guarantee good results on a given application. This analysis is performed in multiple steps, from shrinking the number of studied schedulers to only the best ones, to isolating features that best represent these best schedulers.

5.3.3 *Stability Analysis*

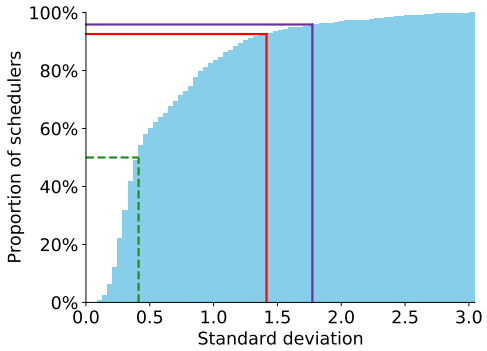
The goal of the stability analysis was to ascertain the statistical validity of our results. Instability can come in multiple flavors: *(i)* a scheduler is unstable across multiple runs of a given application, *(ii)* a scheduler is unstable whatever the application.

To evaluate the stability of a scheduler for a given application, we compute the standard deviation of the 10 runs of each scheduler (σ_s for the scheduler s). We then compute the median (m_σ) and standard deviation (σ_σ) of these standard deviations to define a threshold τ, and build the set of stable schedulers S and the set of unstable schedulers U as follows:

$$\tau = m_\sigma + 2\sigma_\sigma \qquad S = \{s, \sigma_s < \tau\} \qquad U = \{s, \sigma_s \geq \tau\}$$



(a) `bodytrack` **(b)** `nutchindexing`



(c) x264

Figure 5.6: Cumulative distribution of the standard deviation of generated schedulers.

The factor in the definition of τ is determined experimentally to maximize the size of the set of stable schedulers S and obtain good results in subsequent analyses. Figure 5.6 represents cumulative histograms of the standard deviation of our generated schedulers on three applications: `bodytrack`, `johtherippermd5` and `x264`. We also represent the median m_σ, the threshold τ and the standard deviation with CFS as a reference.

STABILITY OF GENERATED SCHEDULERS. Overall, most generated schedulers are stable for all applications. As shown with `bodytrack` and `nutchindexing` in Figures 5.6a and 5.6b, most schedulers have a standard deviation below the τ threshold. For subsequent analyses, we will only use the schedulers present in S and exclude the ones present in U because of their instability. Since the number of schedulers in U is negligible compared to the number of schedulers of S, we will not lose much information. When analyzing the impact of each scheduler feature in Section 5.3.4, only schedulers present in S will be used.

STABILITY OF CFS. Regarding the stability of CFS, we note that for 5 out of 20 applications (`fluidanimate`, `johntherippermd5`, `memory`, `nutchindexing` and `x264`), CFS is less stable than the majority of the generated schedulers (i. e. $\sigma_{CFS} > m_\sigma$). Worse, for `x264`, CFS is considered unstable because its standard deviation is higher than the threshold τ, as shown in Figure 5.6c. This means that some applications, e. g. `x264` in our case, are qualified as unstable in general when it is actually the scheduler that renders them unstable.

UNSTABLE FEATURES. With our feature model and our experimental results, we also wish to study the impact of given features on the stability of the generated schedulers. The simplest way to do this is to look for similarities among the schedulers of the U sets of all applications. Unfortunately, with the current state of our model, all features are fairly distributed among the unstable schedulers. This means that no feature induces instability by itself. We presume that instability is due to combinations of features. However, we were not able to find enough evidence with our data set to consider this presumption a certainty.

5.3.4 *Feature Impact Analysis*

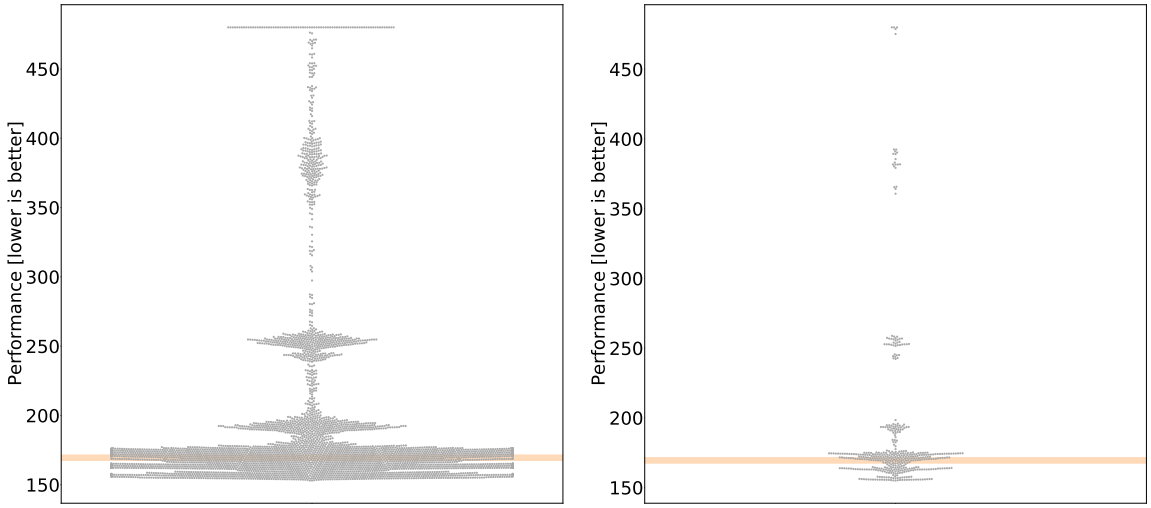
With the results of our stability analysis, we can now analyze the impact of each feature on performance. The general idea is to extract, for a given application, the *good* and the *bad* features. Good features have a high probability of improving the performance of the application while bad features have a low probability of doing so.

The feature impact analysis is split into six phases: data processing, finding the best schedulers, isolating the best features, constructing the scheduler frame, validating the frame and refinement of the frame. At the end, we will have a set of rules regarding features that define what a good scheduler is for a given application, as well as metrics that evaluate the quality of these rules.

We now present the methodology to find these features and apply it on the results of the `facesim` application as an example.

5.3.4.1 *Data Preprocessing*

Before analyzing our experimental data, we must first discard the unstable schedulers, i. e. schedulers in the U set. Due to their instability, these schedulers have no statistical significance when evaluating the performance of a feature. Once these schedulers are discarded, we can simplify the problem by merging all runs of a scheduler into a single value instead of ten: the mean of the ten runs. This simplification is correct since the remaining schedulers are stable, i. e. they have a negligible variability. Let A be this set of schedulers.



(a) Raw data set.

(b) Pruned data set.

Figure 5.7: Preprocessing phase with the `facesim` data set. The orange background spans the results of CFS. (lower is better)

EXAMPLE. Figure 5.7 depicts the preprocessing phase of the methodology in the form of swarm plots. Figure 5.7a shows all executions with all schedulers: there are 4,860 points.[51] The orange background represents the range of results with CFS. We can observe that the performance of our generated schedulers spans a large range of values, with a large number outperforming CFS.

[51] We have 486 schedulers and each is run 10 times.

Figure 5.7b shows the data set after removing unstable schedulers and reducing each scheduler to a single point. In this example, 30 schedulers were removed due to their instability (6.17% of the total). After reducing each scheduler to its mean performance, only 456 points remain from the original 4,860. Despite this reduction, the overall shape of this swarm plot is the same as with the raw data set. This confirms that we did not lose significant information with this preprocessing phase.

5.3.4.2 *Finding the Best Schedulers*

Good and bad features are defined with regards to their probability of improving the application performance. We therefore put our focus on the schedulers that achieve the best performance. We first select a subset of the schedulers in A that are close to the best performance achieved.

Let $best_A$ be the performance of the best scheduler for a given application. We arbitrarily decide that a scheduler "close" to $best_A$ has at most a 10% difference in performance. Let P be the subset of schedulers close to the best, defined as:

$$P = \{x \in A, |x - best_A| \leq 0.1\ best_A\}$$

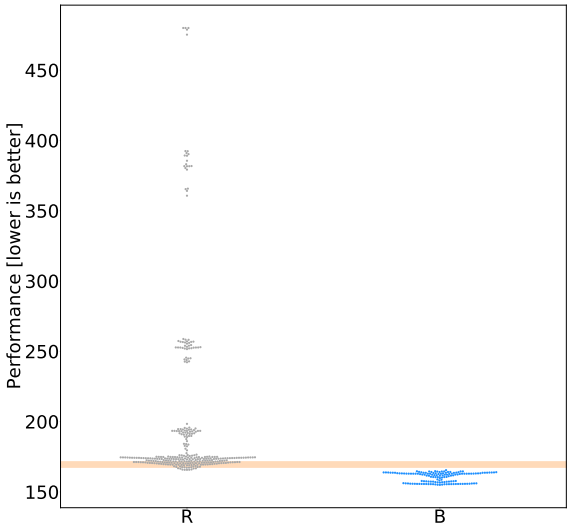


Figure 5.8: Finding the best schedulers for the `facesim` data set. The orange background spans the results of CFS. (lower is better)

However, this is not enough to extract a compact subset of schedulers. With this definition of P, on some experiments, the standard deviation among schedulers of P is high. This means that schedulers are spread across the 10% performance range. We wish to refine our set of best schedulers and only keep the ones closest to the best while accounting for the standard deviation of P (σ_P). We build the set of best schedulers B such as:

$$B = \{x \in P, |x - best_A| \leq |worst_P - \sigma_P|\}$$

With this formula, the higher the standard deviation σ_P, the fewer schedulers we keep in B.

EXAMPLE. We now build the subsets of best schedulers P and B from the preprocessed data set A. The best scheduler for `facesim`, i. e. the one with the lowest execution time, completed in 155 seconds. We therefore have $best_A = 155$. The set of schedulers at most 10% away from $best_A$ in terms of performance, P, contains all values with a performance between $best_A = 155$ and $best_A + 10\% = 170.5$. This set contains 197 schedulers, which amounts to 40.53% of all tested schedulers.

In order to build the set B, we compute the standard deviation of P, $\sigma_P = 4.79$. We then exclude the schedulers that are too far away from $best_A$ while accounting for σ_P and build B. The new set B contains 140 schedulers, which amounts to 28.81% of the total. Figure 5.8 depicts this selection. The blue swarm represents schedulers in B while the gray swarm represents the remaining schedulers. In this example, all schedulers in B outperform CFS, depicted by the orange background.

5.3.4.3 *Isolating the Best Features*

With the set of best schedulers B extracted, we can now select the features with a positive impact on the performance of the application. We do this by counting the number of occurrences of each feature in B. If a feature is present in more than 80% of the schedulers in B, we deem it to be a **good feature**. If it is present in less than 20% of the schedulers in B, it is deemed to be a **bad feature**.

For example, an alternative feature such as the *load metric* has three implementations: *nrRun*, *nrRunBlock* and *usedTime*. If we had a distribution in B of 45–10–45% for each feature, that would mean that *nrRunBlock* is a bad feature and the other two are neither good nor bad. Note that with our thresholds, if a feature is deemed to be good, all other alternative features are necessarily deemed to be bad. We call the set of good and bad features a **scheduler frame**.

EXAMPLE. We count the number of occurrences of each feature in the subset of schedulers B. Table 5.2 summarizes the results. For the `facesim` data set, only one feature, *idle*, has been categorized as a good feature. All other features are roughly equally represented in B. Therefore, the resulting scheduler frame only contains a single feature, *idle*.

5.3.4.4 *Scheduler Frame*

From this scheduler frame representing the good and bad features common to the best schedulers (B), we can categorize our schedulers more precisely. Let F be the subset of schedulers in A that fit in the scheduler frame. We group the best schedulers that fit in the frame as the set FB. The remaining schedulers that are neither best nor fit in the scheduler frame are in the category R such that:

$$FB = F \cap B \qquad\qquad R = A \setminus (F \cup B)$$

We now have four categories that encompass all schedulers of A: FB, F, R and B.

- FB: schedulers fitting the frame among the best schedulers
- F: remaining schedulers fitting the frame
- B: remaining schedulers among the best schedulers
- R: remain schedulers

EXAMPLE. With the resulting scheduler frame, we are now able to build the sets of schedulers F and FB. The set F that represents schedulers fitting into the frame contains 236 schedulers, which amounts to 48.56% of all schedulers. Among the best schedulers (B), 124 fit into

Election	RBtree 44 (31.43%)	Linked list 48 (34.29%)	FIFO 48 (34.29%)
Time slice	Infinite 47 (33.57%)	Fixed 46 (32.86%)	Split 47 (33.57%)
Load metric	nrRun 54 (38.57%)	nrRunBlock 33 (23.57%)	usedTime 53 (37.86%)
Placement distance	SMT 33 (23.57%)	LLC 54 (38.57%)	all 53 (37.86%)
Executor	all 36 (25.71%)	node 51 (36.43%)	core 53 (37.86%)
Idle	no 16 (11.43%)	yes 124 (88.57%)	

Table 5.2: Number of occurrences of each feature in B for the `facesim` data set. Good features are in green, bad features in red.

the scheduler frame and are placed into the set FB, which amounts to 25.51% of all schedulers.

Figure 5.9a shows the distribution among the four sets FB, F, R and B. To improve clarity, schedulers in FB are not depicted in F and B and vice versa. Therefore, each scheduler only appears once. Graphically, we observe that most best schedulers seem to fit in the frame since most schedulers in B are now in FB. This indicates a high representativeness. However, we also observe that F contains a lot of bad schedulers that fit the frame. This indicates that our frame might not be very precise.

5.3.4.5 *Validation*

We need to evaluate the quality of the scheduler frames we build. We do this by measuring two metrics: *representativeness* and *precision*.

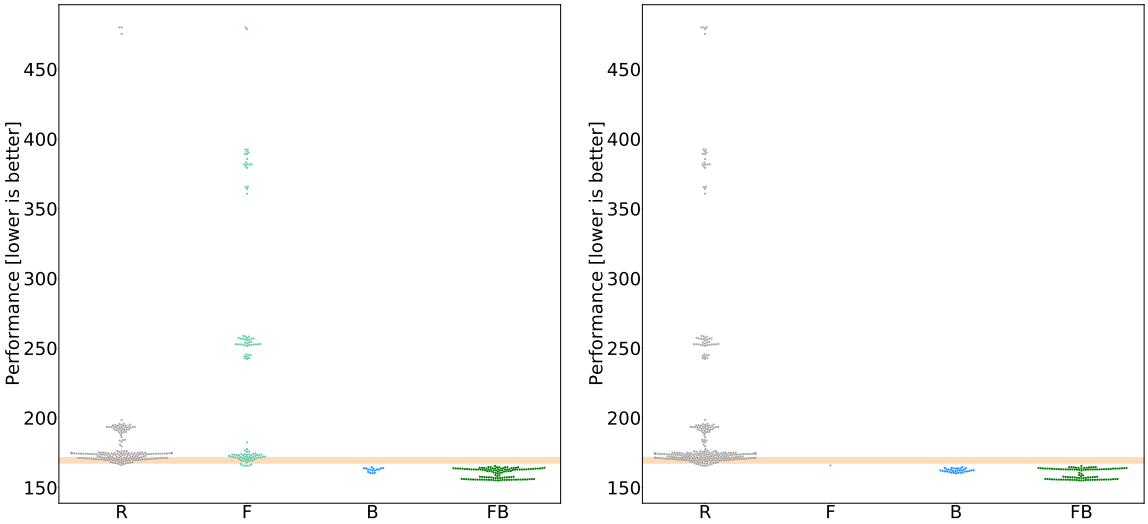
Representativeness ($\mathcal{R}$) is the percentage of schedulers fitting the scheduler frame among the schedulers of B:

$$\mathcal{R} = \frac{|FB|}{|B|}$$

This metric allows us to evaluate if the best schedulers, i. e. schedulers in B, indeed implement the features described by the scheduler frame.

Precision ($\mathcal{P}$) is the percentage of schedulers fitting the scheduler frame in B among all the schedulers fitting the frame (F):

$$\mathcal{P} = \frac{|FB|}{|F|}$$



(a) With initial scheduler frame.

(b) After scheduler frame refinement.

Figure 5.9: Distribution of schedulers among the four sets FB, F, R and B for the `facesim` data set. The orange background spans the results of CFS. (lower is better)

This metric allows us to quantify the number of false positives, i. e. the schedulers that fit in the frame but do not provide good performance. For both metrics, the higher the value the better.

EXAMPLE. To validate our observations, we must compute the representativeness $\mathcal{R}$ and the precision $\mathcal{P}$ of our scheduler frame. With the `facesim` data set, we obtain the following values:

$$\mathcal{R} = 88.57\% \qquad \mathcal{P} = 52.54\%$$

As observed graphically in Figure 5.9a, we obtain a high representativeness and a low precision. This application is a good candidate for the scheduler frame refinement phase.

5.3.4.6 *Scheduler Frame Refinement*

Our objective is to extract the features that make a scheduler good for a given application. Therefore, having too many false positives defeats our purpose. Precision is of the utmost importance and should be maximized, even at the expense of representativeness. When precision is too low, we need to refine the scheduler frame to improve it. For the moment, we consider that below 80%, precision is insufficient.

The refinement we propose slightly differs from our strategy to isolate the best features. We iteratively constrain the frame until precision reaches the 80% threshold. At each iteration, we compute, for each feature, the ratio of its number of occurrences in FB compared to its number of occurrences in F. This ratio represents the potential of a

feature to be a false positive. The feature with the lowest ratio is added to the scheduler frame as a bad feature, and the set F is recomputed with the new frame.

EXAMPLE. For this phase, our starting scheduler frame contains only one rule: *idle* must be in the scheduler. After the first iteration of the refinement process, the feature with the highest potential of being a false positive is the load metric *nrRunBlock*. The new scheduler frame contains two rules: *idle* must be in the scheduler and *nrRunBlock* must not. This first iteration does not give this frame a precision greater than 80%, so we run the algorithm again. After the second iteration, we add a new rule to the scheduler frame: the placement distance *SMT* must not be in the scheduler. We now have a frame with three rules and a precision of 99.07%. The representativeness, however, dropped from 88.57% to 76.43%.

Figure 5.9b depicts the new sets FB, F, R and B after the refinement phase. We observe that F only contains a single scheduler, accordingly to the very high precision of 99.07% we have after refinement. The set FB also shrunk a little as compared to before the refinement (Figure 5.9a), with schedulers returning to the B set. Again, this is consistent to the lower value of the representativeness. Overall, after refinement, we have a scheduler frame that precisely defines good schedulers for the `facesim` application. Having the features in the frame seems to guarantee good performance. However, the converse is not true: not fitting the frame does not mean having bad performance, as shown by the schedulers remaining in B.

5.3.4.7 *Conclusion*

Our methodology enables us to define, for a given application on a given system, the set of scheduler features that work best. We are then able to evaluate the quality of this set of features we call a **scheduler frame** in terms of precision and representativeness. With these two values, we can then refine our scheduler frame to optimize its quality.

We apply our methodology on the data collected from an application from the PARSEC benchmark suite, `facesim`. We show that with our methodology, we are able to find a set of three rules that precisely represent the schedulers that perform best on this application.

In addition to a better understanding of the impact of features on the performance of applications, the scheduler frame could be used for different purposes in the future. One major possibility is to use it in the training set of machine learning tools in order to automatically create the best scheduler possible for a given application. This is made possible thanks to our modular implementation of our feature model.

5.4 FINDING THE BEST SCHEDULER FOR AN APPLICATION

Another possible use of the collection of schedulers we generated is to find the best scheduler for a given application. When this scheduler is found, the `SaaKM` API presented in Section 3.4 can be leveraged to use this scheduler each time the application is run. For example, in cloud environments where a large number of different applications are executed on the same machine, we could always use the best scheduler for every application by changing it on the fly to match the currently running application.

Finding this best scheduler can however be a long process. We present two possible approaches to do this: **brute-force** search and **performance-driven** search. This part is still a work in progress, so we will not present any experimental results.

5.4.1 *Brute-force Search*

This first approach is straightforward: run every generated scheduler and pick the one exhibiting the best performance results. This approach has the advantage of always producing the best possible scheduler among the ones tested. However, the large number of schedulers to test is an obstacle to such an approach.

For example, in our model, we have 486 generated schedulers with our subset of features. Using the brute-force search, users would have to run their application at least 486 times. For applications with long execution times, this is impractical. Add the fact that multiple runs must be performed to have confidence in the results of the search, and this approach becomes completely unusable. To give an idea, running all applications 10 times with all 486 schedulers, plus CFS and CFS with pinned threads, took 1,925 hours (80 days) of machine time.

The growth of the model also makes the number of schedulers, and thus the number of experiments, grow exponentially. Adding a single feature doubles the number of generated schedulers, consequently doubling the experimentation time. Brute-force search is therefore only tractable for users that are able to distribute the search across multiple machines and accept to temporarily disrupt the performance of their application during the process.

5.4.2 *Performance-driven Feature Search*

Bypassing the limitations of the brute-force approach requires to drastically reduce the number of runs needed to find the best scheduler. For statistical significance reasons, reducing the number of runs per scheduler is not a viable option. The only parameter we can work with is the number of schedulers we test. One way to do this is to drive the search depending on the results of the schedulers already tested.

Thanks to our model, features are compartmentalized and can be tested individually. With a few runs of a small number of schedulers, we can start to decide whether a feature has a positive or a negative impact on the performance of the tested application. With this information, we can avoid exploring the complete space of possible schedulers and only focus on a subset of schedulers. We can do this by leveraging the methodology presented in Section 5.3.4.

This type of strategy enables users to rapidly find a satisfying solution to their problem, i. e. a good scheduler for their application. However, by not testing all schedulers, we might miss the actual best scheduler. We can only find a good solution, not necessarily the best one.

This performance-driven search could be furthermore enhanced by learning the impact of individual features with certain types of workloads. When testing schedulers, we could collect, in addition to performance metrics, metadata that allows characterization of the workload, e. g. number of threads, I/O- or CPU-bounded application, ... Using all this data, we could build a model that predicts the performance of given features on a workload based on previous experiments. There exists multiple machine learning tools adapted for such problems. This is currently a work-in-progress.

5.5 CONCLUSION

Analyzing the performance of a scheduler is difficult. More precisely, evaluating each feature of a scheduler is challenging because of the way schedulers are written. We show that the features of CFS are too intertwined to allow for an evaluation of each one individually.

To get around this problem, we propose a model of a scheduler in the form of a **feature model**. This model describes a scheduler as a set of features linked by dependency rules. We implement each feature in isolation so that they can be used interchangeably. Thanks to `SaaKM`, we then execute a large number of applications on the 486 schedulers we generate from our model.

With the results of our experiments, we analyze the stability of the generated schedulers as well as of CFS. We show that some applications were thought to be unstable when the instability came from CFS and not the application.

We also propose a methodology to evaluate the performance of each feature individually on a given application. With this methodology, we are able to build a **scheduler frame** that describes the set of desirable and undesirable features for an application.

Finally, we propose two approaches for end users to build an **application-specific scheduler** easily. This part is an ongoing work.

FUTURE WORK. The current model is limited, with less than twenty implemented features. We wish to extend it and add more features from existing schedulers, as well as novel features.

We also wish to apply our feature evaluation methodology on more applications, with more features. This would allow us to evaluate our ability to use this data as a training set for our approaches to build application-specific schedulers.

Finally, we are currently working on the performance-driven feature search describe in Section 5.4.2. The two main areas of research are the machine learning approach as well as correlating application behavior to features.

RELATED PUBLICATION. The work described in this chapter has been the subject of a publication in a national conference [68].

6

CONCLUSION

In the last couple decades, increasing the processing power of computers is a synonym for increasing the number of computing units. Multi-cores took over the CPU market in a few years. With the multiplication of computing resources comes a greater need to correctly allocate threads to cores, a job carried out by the scheduler. However, improper scheduling incurs poor resource utilization and therefore subpar performance. Ultimately, this leads to user disappointment. Thread schedulers are at the center of the performance of modern systems.

Developing a thread scheduler is a daunting task. The dazzling evolution of the hardware lead to an increased complexity in its understanding. Software applications, on the other hand, are more and more diverse, leading to a large number of different performance requirements. General-purpose OSs schedulers struggle to be as generic as possible and fail to succeed in this illusory enterprise.

In this thesis, we studied a variety of existing schedulers and pointed out three axes of improvement: **scheduler development**, **performance enhancement** and **application-specific schedulers**. For each axis, we provide contributions that will help scheduler developers produce new schedulers and end users get the best performance for their applications.

6.1 SCHEDULER DEVELOPMENT

CONTRIBUTIONS. We propose **Ipanema**, a domain-specific language (DSL) that eases the development of new schedulers with a focus on code correctness and the possibility to formally verify properties on the scheduling algorithm. Alongside the DSL, we provide a complete tool chain with a modified Linux kernel that features scheduler hot-plugging through the `SaaKM` kernel API and a compiler that generates efficient kernel modules in C and proofs in WhyML.

Thanks to the expressiveness of our DSL, we write scheduling policies inspired by CFS and ULE with a smaller code footprint. The CFS-like policies are 5.7× smaller than the original and the ULE-like policies are 2.5× smaller than their original counterpart. Among those policies, some are also proven work-conserving.

We evaluate the generated C code of our policies on a set of applications including compilation, databases and HPC. Overall, our policies perform similarly or better than the vanilla CFS, comforting us in the idea that we can produce schedulers that are compact, efficient and verified in Ipanema.

FUTURE WORK. We propose two areas of improvement for this work. The first one consists of extending the proof library with new properties such as thread liveness or freedom from frequency inversions. The second area focuses on faithfully reproducing existing production schedulers in Ipanema in order to formally verify if they are algorithmically correct. This may lead to the discovery of unknown bugs in schedulers like CFS or ULE, and attempts to fix them. However, doing this would require to add new features to the Ipanema language and standard library. For example, new data structures for thread storage will be needed, as well to new methods that allow developers to query the architectural state of cores, e. g. frequency.

At a higher level, we also think that interesting work could be done on a meta-scheduler. With our `SaaKM` API, multiple scheduling policies can live in the kernel. As with Linux scheduling classes, we only implement a static priority list to arbitrate between policies. In the future, we wish to be able to change the arbitration between scheduling policies. This could be done through an extension of the abstractions defined in the Ipanema DSL.

6.2 PERFORMANCE ENHANCEMENT

CONTRIBUTIONS. In order to enhance performance, we implement a set of monitoring and visualization tools that record events in the kernel at a high resolution and display them in a scalable and scriptable graphical interface. Thanks to these tools, we discover a novel problem on modern processors, **frequency inversion**. On CPUs featuring per-core dynamic voltage and frequency scaling (DVFS), we can have situations when idle cores run at high frequencies while busy cores run at low frequencies. This mismatch between load and frequency is due to the long frequency transition latencies (FTLs) of processors and to the scheduling algorithms of general-purpose OSs that do not correctly account for the frequency of cores. We give a detailed analysis of this problem and propose two solutions for Linux. Our best solution, S_{move} provides up to 56% improvement and a worst deterioration of 8.4% on a set of 60 diverse applications we evaluate. Overall, we significantly improve the performance on a large number of applications and do not significantly degrade the performance of applications not disturbed by frequency inversions. The S_{move} solution has been submitted to the Linux kernel community.

In addition to these enhancements, we provide a detailed analysis of the FTL for multiple CPUs as well as a methodology to measure it. We show that the frequency scaling algorithm largely depends on the CPU model and targeted market. We also show that frequency scaling is all but instantaneous, and should be accounted for when designing schedulers.

FUTURE WORK. With our work on frequency inversions, we believe the schedulers should account for the frequency of individual cores not treat them as equals. However, the decisions of the scheduler can also have an impact on the frequency of cores. We need an accurate model of the frequency behavior of the CPU to predict this impact and act accordingly. This model should account for multiple parameters that impact the frequency such as the number of active cores, the FTL, temperature and instruction set.

Another possible way of solving frequency issues regarding the scheduler would be to improve scaling governors and reduce the duration of hardware reconfiguration. Indeed, in a perfect setup, the FTL would be instantaneous, and most problems regarding frequency would disappear, starting with frequency inversions.

6.3 APPLICATION-SPECIFIC SCHEDULERS

CONTRIBUTIONS. From the expertise we accumulate in our work, we define a **feature model** representing a scheduler. From this model, we are able to implement each feature independently. We can then use any feature to build modular schedulers, in accordance with the rules of the model.

In addition to the model, we propose multiple methodologies to evaluate the generated schedulers and their features individually. We propose a stability analysis that helped us discover that some application known as unstable were not intrinsically unstable: it was CFS that created this instability.

We also propose a methodology to evaluate features individually and extract a set of good and bad features for a given application. We call this set a **scheduler frame**.

Finally, thanks to the modular implementation of our feature model, we can start designing **application-specific schedulers** in an automatic manner. We propose two approaches to do so. The first one is a brute-force approach allowed by the modular implementation of our feature model that allows us to build all possible combinations. The second one aims at converging towards an efficient scheduler without testing all combinations of features. The evaluation methodology proposed can be used to guide this search.

FUTURE WORK. In order to improve the usability of our methodology, we need to extend the number of features we implement. With a larger number of features, we could craft more specific schedulers for applications.

Improving the performance-driven feature search is also essential. In our opinion, the most promising approach is to use machine learning techniques. A model would be trained with a large number of applications exhibiting various application patterns. With this trained model, we could quickly find a well-suited scheduler for a given new application with only a few runs to determine its behavior. The training set used would also include the scheduler frames we introduced.

PUBLICATIONS

During this thesis, countless papers were submitted in international and national conferences and workshops. Here is the list of the ones that were accepted.

INTERNATIONAL CONFERENCE PAPERS

- Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, Nicolas Palix, Redha Gouicem, Julien Sopena, Julia Lawall, and Gilles Muller. "Towards Proving Optimistic Multicore Schedulers." In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*. Ed. by Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal. ACM, 2017, pp. 18–23. DOI: 10.1145/3102980.3102984. URL: https://doi.org/10.1145/3102980.3102984.
- Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. "The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS." In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. by Haryadi S. Gunawi and Benjamin Reed. USENIX Association, 2018, pp. 85–96. URL: https://www.usenix.org/conference/atc18/presentation/bouron.
- Damien Carver, Redha Gouicem, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. "Fork/Wait and Multicore Frequency Scaling: a Generational Clash." In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 2019, pp. 53–59. DOI: 10.1145/3365137.3365400. URL: https://doi.org/10.1145/3365137.3365400.
- Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. "Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance." In: *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. Ed. by Ada Gavrilovska and Erez Zadok. USENIX Association, 2020, pp. 435–448. URL: https://www.usenix.org/conference/atc20/presentation/gouicern.

- Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. "Provable multicore schedulers with Ipanema: application to work conservation." In: *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. Ed. by Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer. ACM, Apr. 2020, 3:1–3:16. DOI: 10.1145/3342195.3387544. URL: https://doi.org/10.1145/3342195.3387544.

NATIONAL CONFERENCE PAPERS

- Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, and Nicolas Palix. "Ipanema : un Langage Dédié pour le Développement d'Ordonnanceurs Multi-Coeur Sûrs." In: *Compas 2017: Conférence d'informatique en Parallélisme, Architecture et Système*. Sophia Antipolis, France, June 2017. URL: https://hal.sorbonne-universite.fr/hal-02111160.
- Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, and Nicolas Palix. "Understanding Scheduler Performance : a Feature-Based Approach." In: *Compas 2019: Conférence d'informatique en Parallélisme, Architecture et Système*. Anglet, France, June 2019. URL: https://hal.archives-ouvertes.fr/hal-02558763.

PRODUCED SOFTWARE

LINUX KERNEL DEVELOPMENT

- Ipanema-enabled kernel, with the `SaaKM` scheduling class:
  `https://gitlab.inria.fr/ipanema-public/ipanema-kernel`
- Frequency inversion patches (S_{local} and S_{move}):
  `https://gitlab.inria.fr/whisper-public/atc20`

FREQUENCY-RELATED TOOLS

- `frequency_logger`:
  `https://github.com/rgouicem/frequency_logger`
- Per-core DVFS tester:
  `https://github.com/rgouicem/percoreDVFStester`

VISUALIZATION TOOLS

- SchedDisplay:
  `https://github.com/carverdamien/SchedDisplay`

BIBLIOGRAPHY

[1] Luca Abeni, Alessandro Biondi, and Enrico Bini. "Hierarchical scheduling of real-time tasks over Linux-based virtual machines." In: *J. Syst. Softw.* 149 (2019), pp. 234–249. DOI: 10.1016/j.jss.2018.12.008. URL: https://doi.org/10.1016/j.jss.2018.12.008 (cit. on p. 40).

[2] Luca Abeni and Giorgio C. Buttazzo. "Integrating Multimedia Applications in Hard Real-Time Systems." In: *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*. 1998, pp. 4–13. DOI: 10.1109/REAL.1998.739726. URL: https://doi.org/10.1109/REAL.1998.739726 (cit. on p. 31).

[3] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. "Analysis of a Reservation-Based Feedback Scheduler." In: *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), Austin, Texas, USA, December 3-5, 2002*. IEEE Computer Society, 2002, pp. 71–80. DOI: 10.1109/REAL.2002.1181563. URL: https://doi.org/10.1109/REAL.2002.1181563 (cit. on p. 16).

[4] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby C. Murray, Gerwin Klein, and Gernot Heiser. "CoGENT: Verifying High-Assurance File System Implementations." In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*. 2016, pp. 175–188. DOI: 10.1145/2872362.2872404. URL: https://doi.org/10.1145/2872362.2872404 (cit. on p. 45).

[5] AMD. *AMD - Turbo Core Technology*. URL: https://www.amd.com/en/technologies/turbo-core (cit. on p. 80).

[6] AMD. *AMD Ryzen™ Technology: Precision Boost 2 Performance Enhancement*. URL: https://www.amd.com/en/support/kb/faq/cpu-pb2 (cit. on p. 80).

[7] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. "Real-Time Scheduling on Multicore Platforms." In: *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*. IEEE Computer Society, 2006, pp. 179–190. DOI: 10.1109/RTAS.2006.35. URL: https://doi.org/10.1109/RTAS.2006.35 (cit. on p. 16).

[8] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism." In: *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*. Ed. by Henry M. Levy. ACM, 1991, pp. 95–109. DOI: 10.1145/121132.121151. URL: https://doi.org/10.1145/121132.121151 (cit. on p. 37).

[9] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. "Scheduling Algorithms with Bus Bandwidth Considerations for SMPs." In: *32nd International Conference on Parallel Processing (ICPP 2003), 6-9 October 2003, Kaohsiung, Taiwan*. IEEE Computer Society, 2003, pp. 547–554. DOI: 10.1109/ICPP.2003.1240622. URL: https://doi.org/10.1109/ICPP.2003.1240622 (cit. on p. 16).

[10] *Apache Hadoop*. URL: https://hadoop.apache.org/ (cit. on p. 116).

[11] *ARM - Technologies - ARM big.LITTLE*. URL: https://www.arm.com/why-arm/technologies/big-little (cit. on p. 8).

[12] Neil C. Audsley. *Deadline Monotonic Scheduling*. 1990 (cit. on p. 16).

[13] David H. Bailey, Eric Barszcz, John T. Barton, D. S. Browning, Robert L. Carter, Leonardo Dagum, Rod Fatoohi, Paul O. Frederickson, T. A. Lasinski, Robert Schreiber, Horst D. Simon, V. Venkatakrishnan, and Sisira Weeratunga. "The NAS parallel benchmarks - summary and preliminary results." In: *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*. 1991, pp. 158–165. DOI: 10.1145/125826.125925. URL: https://doi.org/10.1145/125826.125925 (cit. on pp. 62, 88).

[14] Scott A. Banachowski and Scott A. Brandt. "Better Real-Time Response for Time-Share Scheduling." In: *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 2003, p. 124. DOI: 10.1109/IPDPS.2003.1213246. URL: https://doi.org/10.1109/IPDPS.2003.1213246 (cit. on p. 16).

[15] Jon C. R. Bennett and Hui Zhang. "WF^{2}Q: Worst-Case Fair Weighted Fair Queueing." In: *Proceedings IEEE INFOCOM '96, The Conference on Computer Communications, Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking the Next Generation, San Francisco, CA, USA, March 24-28, 1996*. IEEE Computer Society, 1996, pp. 120–128. DOI: 10.1109/INFCOM.1996.497885. URL: https://doi.org/10.1109/INFCOM.1996.497885 (cit. on p. 15).

[16] Arnd Bergmann. *Killing the Big Kernel Lock*. 2010. URL: https://lwn.net/Articles/380174/ (cit. on p. 21).

[17] Christian Bienia. "Benchmarking Modern Multiprocessors." PhD thesis. Princeton University, Jan. 2011 (cit. on p. 116).

[18] Antoine Blin, Cédric Courtaud, Julien Sopena, Julia L. Lawall, and Gilles Muller. "Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System." In: *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*. IEEE Computer Society, 2016, pp. 109–119. DOI: 10.1109/ECRTS.2016.18. URL: https://doi.org/10.1109/ECRTS.2016.18 (cit. on p. 17).

[19] R. T. Blosk. "The Instruction Unit of the Stretch Computer." In: *Papers Presented at the December 13-15, 1960, Eastern Joint IRE-AIEE-ACM Computer Conference*. New York, NY, USA: Association for Computing Machinery, 1960. ISBN: 9781450378710. DOI: 10.1145/1460512.1460539. URL: https://doi.org/10.1145/1460512.1460539 (cit. on p. 7).

[20] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Why3: Shepherd Your Herd of Provers." In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. https://hal.inria.fr/hal-00790310. Wrocław, Poland, Aug. 2011, pp. 53–64 (cit. on pp. 45, 59).

[21] *Bokeh documentation*. URL: https://bokeh.pydata.org/ (cit. on p. 72).

[22] Justinien Bouron. *[PATCH] Fix bug in which the long term ULE load balancer is executed only once*. 2017. URL: https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=223914 (cit. on p. 61).

[23] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. "The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS." In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. by Haryadi S. Gunawi and Benjamin Reed. USENIX Association, 2018, pp. 85–96. URL: https://www.usenix.org/conference/atc18/presentation/bouron (cit. on pp. 35, 105).

[24] *bpftrace: High-level tracing language for Linux eBPF*. URL: https://github.com/iovisor/bpftrace (cit. on p. 71).

[25] Andrey Breslav and Roman Elizarov. *Kotlin Coroutines*. URL: https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md (cit. on p. 39).

[26] Gunther Buschmann, Hans-Thomas Ebner, and Wieland Kuhn. "Electronic Brake Force Distribution Control - A Sophisticated Addition to ABS." In: *International Congress & Exposition*. SAE International, Feb. 1992. DOI: https://doi.org/10.4271/920646. URL: https://doi.org/10.4271/920646 (cit. on p. 13).

[27] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. "The Cascade High Productivity Language." In: *9th International Workshop on High-Level Programming Models and Supportive Environments (HIPS 2004), 26 April 2004, Santa Fe, NM, USA*. IEEE Computer Society, 2004, pp. 52–60. DOI: 10.1109/HIPS.2004.10002. URL: http://doi.ieeecomputersociety.org/10.1109/HIPS.2004.10002 (cit. on p. 38).

[28] Damien Carver, Redha Gouicem, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. "Fork/Wait and Multicore Frequency Scaling: a Generational Clash." In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 2019, pp. 53–59. DOI: 10.1145/3365137.3365400. URL: https://doi.org/10.1145/3365137.3365400 (cit. on p. 104).

[29] *CFS Scheduler Documentation*. URL: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt (cit. on p. 27).

[30] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: an object-oriented approach to non-uniform cluster computing." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. Ed. by Ralph E. Johnson and Richard P. Gabriel. ACM, 2005, pp. 519–538. DOI: 10.1145/1094811.1094852. URL: https://doi.org/10.1145/1094811.1094852 (cit. on p. 38).

[31] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Mert Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying a high-performance crash-safe file system using a tree specification." In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 270–286. DOI: 10.1145/3132747.3132776. URL: https://doi.org/10.1145/3132747.3132776 (cit. on p. 61).

[32] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. "Using Crash Hoare logic for certifying the FSCQ file system." In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*

2015, Monterey, CA, USA, October 4-7, 2015. 2015, pp. 18–37. DOI: 10.1145/2815400.2815402. URL: https://doi.org/10.1145/2815400.2815402 (cit. on p. 61).

[33] Tim Chen, Leonid I Ananiev, and Alexander V Tikhonov. "Keeping kernel performance from regressions." In: *Linux Symposium*. Vol. 1. 2007, pp. 93–102 (cit. on p. 61).

[34] Austin Clements. *runtime: non-cooperative goroutine preemption*. 2018. URL: https://github.com/golang/go/issues/24543 (cit. on p. 38).

[35] Austin Clements. *runtime: clean up async preemption loose ends*. 2020. URL: https://github.com/golang/go/issues/36365 (cit. on p. 38).

[36] *cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages*. URL: https://github.com/AlDanial/cloc (cit. on p. 106).

[37] Jonathan Corbet. *The real BKL end game*. 2011. URL: https://lwn.net/Articles/424657/ (cit. on p. 21).

[38] Jonathan Corbet. *Per-entity load tracking*. Jan. 2013. URL: https://lwn.net/Articles/531853/ (cit. on p. 28).

[39] Jonathan Corbet. *TurboSched: the return of small-task packing*. 2019. URL: https://lwn.net/Articles/792471/ (cit. on p. 103).

[40] Timothy Creech, Aparna Kotha, and Rajeev Barua. "Efficient multiprogramming for multicores with SCAF." In: *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*. Ed. by Matthew K. Farrens and Christos Kozyrakis. ACM, 2013, pp. 334–345. DOI: 10.1145/2540708.2540737. URL: https://doi.org/10.1145/2540708.2540737 (cit. on p. 39).

[41] *Crystal Language Reference - Concurrency*. URL: https://crystal-lang.org/reference/guides/concurrency.html (cit. on p. 38).

[42] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. "Self-tuning schedulers for legacy real-time applications." In: *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*. Ed. by Christine Morin and Gilles Muller. ACM, 2010, pp. 55–68. DOI: 10.1145/1755913.1755921. URL: https://doi.org/10.1145/1755913.1755921 (cit. on p. 16).

[43] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. "Parallel computer architecture - a hardware / software approach." In: Morgan Kaufmann, 1999, p. 32. ISBN: 978-1-55860-343-1 (cit. on p. 9).

[44] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55 (cit. on pp. 17, 39).

[45] *Datashader documentation*. URL: https://datashader.org/ (cit. on p. 72).

[46] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. "Analysis and Simulation of a Fair Queueing Algorithm." In: *SIGCOMM '89, Proceedings of the ACM Symposium on Communications Architectures & Protocols, Austin, TX, USA, September 19-22, 1989*. Ed. by Lawrence H. Landweber. ACM, 1989, pp. 1–12. DOI: 10.1145/75246.75248. URL: https://doi.org/10.1145/75246.75248 (cit. on p. 15).

[47] John DiGiglio, David Hunt, Ai Bee Lim, Chris MacNamara, and Timothy Miskell. *Intel® Speed Select Technology – Base Frequency - Enhancing Performance*. 2019. URL: https://builders.intel.com/docs/networkbuilders/intel-speed-select-technology-base-frequency-enhancing-performance.pdf (cit. on p. 79).

[48] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. 1st. Addison-Wesley Professional, 2015. ISBN: 0134190440 (cit. on p. 17).

[49] Michael Drescher, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. "A flattened hierarchical scheduler for real-time virtualization." In: *2016 International Conference on Embedded Software, EMSOFT 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. Ed. by Petru Eles and Rahul Mangharam. ACM, 2016, 12:1–12:10. DOI: 10.1145/2968478.2968501. URL: https://doi.org/10.1145/2968478.2968501 (cit. on p. 40).

[50] Kenneth J. Duda and David R. Cheriton. "Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose schedular." In: *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999*. Ed. by David Kotz and John Wilkes. ACM, 1999, pp. 261–276. DOI: 10.1145/319151.319169. URL: https://doi.org/10.1145/319151.319169 (cit. on p. 16).

[51] Adam Dunkels. *Protothreads*. URL: http://dunkels.com/adam/pt/index.html (cit. on p. 39).

[52] *Dyalog Programming Reference Guide - Threads*. URL: http://help.dyalog.com/latest/index.htm#Language/Introduction/Threads/Multithreading%20Overview.htm (cit. on p. 38).

[53] Eclipse. *Trace Compass*. URL: https://www.eclipse.org/tracecompass/ (cit. on p. 71).

[54] *Eclipse OpenJ9 Documentation - Java Dump*. URL: https://www.eclipse.org/openj9/docs/dump_javadump/#threads (cit. on p. 38).

[55] Ralf S. Engelschall. *GNU Pth - The GNU Portable Threads*. URL: https://www.gnu.org/software/pth/ (cit. on p. 39).

[56] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. "A Non-Work-Conserving Operating System Scheduler for SMT Processors." In: *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA (Vol. 33)*. 2006, pp. 10–17 (cit. on p. 16).

[57] Alexandra Fedorova, Margo I. Seltzer, Christopher Small, and Daniel Nussbaum. "Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design." In: *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 395–398. URL: http://www.usenix.org/events/usenix05/tech/general/fedorova.html (cit. on p. 16).

[58] Alexandra Fedorova, Margo I. Seltzer, and Michael D. Smith. "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler." In: *16th International Conference on Parallel Architectures and Compilation Techniques (PACT 2007), Brasov, Romania, September 15-19, 2007*. IEEE Computer Society, 2007, pp. 25–38. DOI: 10.1109/PACT.2007.40. URL: http://doi.ieeecomputersociety.org/10.1109/PACT.2007.40 (cit. on p. 16).

[59] Matt Fleming. *A thorough introduction to eBPF*. 2017. URL: https://lwn.net/Articles/740157/ (cit. on p. 71).

[60] *FreeBSD 12.0 Performance Against Windows & Linux On An Intel Xeon Server*. Dec. 2018. URL: https://www.phoronix.com/scan.php?page=article&item=freebsd-12-windows (cit. on p. 105).

[61] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The Implementation of the Cilk-5 Multithreaded Language." In: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*. Ed. by Jack W. Davidson, Keith D. Cooper, and A. Michael Berman. ACM, 1998, pp. 212–223. DOI: 10.1145/277650.277725. URL: https://doi.org/10.1145/277650.277725 (cit. on p. 39).

[62] *GalliumOS: A fast and lightweight Linux distro for ChromeOS devices*. URL: https://galliumos.org/ (cit. on p. 33).

[63] *Getting Started with Erlang - Concurrent Programming*. URL: https://erlang.org/doc/getting_started/conc_prog.html (cit. on p. 38).

[64] Google. *Google Open Source Blog – Understanding Scheduling Behavior with SchedViz*. 2019. URL: https://opensource.googleblog.com/2019/10/understanding-scheduling-behavior-with.html (cit. on p. 71).

[65] Google. *SchedViz: A tool for gathering and visualizing kernel scheduling traces on Linux machines*. URL: https://github.com/google/schedviz (cit. on p. 71).

[66] Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. "Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance." In: *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. Ed. by Ada Gavrilovska and Erez Zadok. USENIX Association, 2020, pp. 435–448. URL: https://www.usenix.org/conference/atc20/presentation/gouicern (cit. on p. 104).

[67] Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, and Nicolas Palix. "Ipanema : un Langage Dédié pour le Développement d'Ordonnanceurs Multi-Coeur Sûrs." In: *Compas 2017: Conférence d'informatique en Parallélisme, Architecture et Système*. Sophia Antipolis, France, June 2017. URL: https://hal.sorbonne-universite.fr/hal-02111160 (cit. on p. 68).

[68] Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, and Nicolas Palix. "Understanding Scheduler Performance : a Feature-Based Approach." In: *Compas 2019: Conférence d'informatique en Parallélisme, Architecture et Système*. Anglet, France, June 2019. URL: https://hal.archives-ouvertes.fr/hal-02558763 (cit. on p. 130).

[69] graysky. *CPU Schedulers Compared*. URL: http://repo-ck.com/bench/cpu_schedulers_compared.pdf (cit. on p. 33).

[70] *greenlet: Lightweight concurrent programming*. URL: https://greenlet.readthedocs.io/ (cit. on p. 39).

[71] Michael Gschwind, H. Peter Hofstee, Brian K. Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. "Synergistic Processing in Cell's Multicore Architecture." In: *IEEE Micro* 26.2 (2006), pp. 10–24. DOI: 10.1109/MM.2006.41. URL: https://doi.org/10.1109/MM.2006.41 (cit. on p. 8).

[72] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels." In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah,*

GA, USA, November 2-4, 2016. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 653–669. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu (cit. on p. 61).

[73] Leonidas J. Guibas and Robert Sedgewick. "A Dichromatic Framework for Balanced Trees." In: *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. IEEE Computer Society, 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3. URL: https://doi.org/10.1109/SFCS.1978.3 (cit. on pp. 27, 112).

[74] Michael González Harbour, Mark H. Klein, and John P. Lehoczky. "Fixed priority scheduling periodic tasks with varying execution priority." In: *Proceedings of the Real-Time Systems Symposium - 1991, San Antonio, Texas, USA, December 1991*. IEEE Computer Society, 1991, pp. 116–128. DOI: 10.1109/REAL.1991.160365. URL: https://doi.org/10.1109/REAL.1991.160365 (cit. on p. 15).

[75] Tim Harris, Martin Maas, and Virendra J. Marathe. "Callisto: co-scheduling parallel runtime systems." In: *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. Ed. by Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel. ACM, 2014, 24:1–24:14. DOI: 10.1145/2592798.2592807. URL: https://doi.org/10.1145/2592798.2592807 (cit. on p. 39).

[76] Gary J. Henry. "The UNIX system: The fair share scheduler." In: *AT&T Bell Lab. Tech. J.* 63.8 (1984), pp. 1845–1857. DOI: 10.1002/j.1538-7305.1984.tb00068.x. URL: https://doi.org/10.1002/j.1538-7305.1984.tb00068.x (cit. on p. 15).

[77] *HotSpot Runtime Overview*. URL: https://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html (cit. on p. 38).

[78] Brett Howse. *Examining Intel's New Speed Shift Tech on Skylake: More Responsive Processors*. Nov. 2015. URL: https://www.anandtech.com/show/9751/examining-intel-skylake-speed-shift-more-responsive-processors (cit. on p. 79).

[79] Galen C. Hunt and James R. Larus. "Singularity: rethinking the software stack." In: *Operating Systems Review* 41.2 (2007), pp. 37–49. DOI: 10.1145/1243418.1243424. URL: https://doi.org/10.1145/1243418.1243424 (cit. on p. 17).

[80] Wenwei W. Hwu and Yale N. Patt. "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality." In: *Proceedings of the 13th Annual Symposium on Computer Architecture, Tokyo, Japan, June 1986*. Ed. by Hideo Aiso. IEEE Computer Society, 1986, pp. 297–306. URL: https://dl.acm.org/citation.cfm?id=17391 (cit. on p. 7).

[81] *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications*. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), Jan. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html (cit. on p. 18).

[82] Intel. *Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor*. Mar. 2004. URL: http://download.intel.com/design/network/papers/30117401.pdf (cit. on p. 79).

[83] Intel. *Intel® Turbo Boost Technology 2.0 - Higher Performance When You Need It Most*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html (cit. on p. 79).

[84] *Intel HiBench - HiBench is a big data benchmark suite*. URL: https://github.com/intel-hadoop/HiBench (cit. on p. 116).

[85] *Intel® Hyper-Threading Technology*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html (cit. on p. 7).

[86] *Introducing AMD Ryzen™ Desktop Processors with Radeon™ Vega Graphics: The Next Step in Computing*. URL: https://www.amd.com/en/partner/amd-ryzen-radeon-vega (cit. on p. 8).

[87] *JamVM*. URL: http://jamvm.sourceforge.net/ (cit. on p. 38).

[88] *JikesRVM - Chapter 14: Core Runtime Services*. URL: https://www.jikesrvm.org/UserGuide/CoreRuntimeServices/index.html#x17-18500014.3 (cit. on p. 38).

[89] *Julia 1.5-DEV Documentation - Control Flow*. URL: https://docs.julialang.org/en/latest/manual/control-flow/#man-tasks-1 (cit. on p. 38).

[90] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. "Profiling a warehouse-scale computer." In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. Ed. by Deborah T. Marr and David H. Albonesi. ACM, 2015, pp. 158–169. DOI: 10.1145/2749469.2750392. URL: https://doi.org/10.1145/2749469.2750392 (cit. on p. 43).

[91] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231 (cit. on p. 110).

[92] Judy Kay and Piers Lauder. "A Fair Share Scheduler." In: *Commun. ACM* 31.1 (1988), pp. 44–55. DOI: 10.1145/35043.35047. URL: https://doi.org/10.1145/35043.35047 (cit. on p. 15).

[93] William Kennedy. *Scheduling In Go : Part II - Go Scheduler*. 2018. URL: https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html (cit. on p. 38).

[94] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: formal verification of an OS kernel." In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596 (cit. on p. 61).

[95] Con Kolivas. *Re: BFS vs. mainline scheduler benchmarks and measurements*. 2009. URL: https://lore.kernel.org/lkml/200909101102.56615.kernel@kolivas.org/ (cit. on p. 69).

[96] Con Kolivas. *FAQs about BFS*. URL: http://ck.kolivas.org/patches/bfs/bfs-faq.txt (cit. on p. 33).

[97] Con Kolivas. *MuQSS - The Multiple Queue Skiplist Scheduler*. URL: http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt (cit. on p. 33).

[98] Alexey Kopytov. *Sysbench - Scriptable database and system performance benchmark*. URL: https://github.com/akopytov/sysbench (cit. on pp. 62, 116).

[99] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. "Heterogeneous Managed Runtime Systems: A Computer Vision Case Study." In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. ACM, 2017, pp. 74–82. DOI: 10.1145/3050748.3050764. URL: https://doi.org/10.1145/3050748.3050764 (cit. on p. 38).

[100] Martin Kováčik. *PostgreSQL benchmark on FreeBSD, CentOS, Ubuntu Debian and openSUSE*. Dec. 2017. URL: https://redbyte.eu/en/blog/postgresql-benchmark-freebsd-centos-ubuntu-debian-opensuse/ (cit. on p. 105).

[101] Charles Krasic, Mayukh Saubhasik, Anirban Sinha, and Ashvin Goel. "Fair and timely scheduling via cooperative polling." In: *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*. Ed. by Wolfgang Schröder-Preikschat, John

Wilkes, and Rebecca Isaacs. ACM, 2009, pp. 103–116. DOI: 10.1145/1519065.1519077. URL: https://doi.org/10.1145/1519065.1519077 (cit. on p. 16).

[102] Sun Labs. *Fortress GitHub repository*. URL: https://github.com/stokito/fortress-lang (cit. on p. 38).

[103] Adam Lackorzynski, Alexander Warg, Marcus Völp, and Hermann Härtig. "Flattening hierarchical scheduling." In: *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*. Ed. by Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr. ACM, 2012, pp. 93–102. DOI: 10.1145/2380356.2380376. URL: https://doi.org/10.1145/2380356.2380376 (cit. on p. 40).

[104] John P. Lehoczky. "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines." In: *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990*. IEEE Computer Society, 1990, pp. 201–209. DOI: 10.1109/REAL.1990.128748. URL: https://doi.org/10.1109/REAL.1990.128748 (cit. on p. 15).

[105] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. "Provable multicore schedulers with Ipanema: application to work conservation." In: *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. Ed. by Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer. ACM, Apr. 2020, 3:1–3:16. DOI: 10.1145/3342195.3387544. URL: https://doi.org/10.1145/3342195.3387544 (cit. on pp. 60, 68).

[106] Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, Nicolas Palix, Redha Gouicem, Julien Sopena, Julia Lawall, and Gilles Muller. "Towards Proving Optimistic Multicore Schedulers." In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*. Ed. by Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal. ACM, 2017, pp. 18–23. DOI: 10.1145/3102980.3102984. URL: https://doi.org/10.1145/3102980.3102984 (cit. on p. 68).

[107] Jochen Liedtke. "On micro-Kernel Construction." In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. Ed. by Michael B. Jones. ACM, 1995, pp. 237–250. DOI: 10.1145/224056.224075. URL: https://doi.org/10.1145/224056.224075 (cit. on p. 17).

[108] *Linux Test Project - Testing Linux, one syscall at a time*. URL: https://linux-test-project.github.io/ (cit. on pp. 61, 98, 116).

[109] John S. Liptay. "Structural Aspects of the System/360 Model 85 II: The Cache." In: *IBM Syst. J.* 7.1 (1968), pp. 15–21. DOI: 10.1147/sj.71.0015. URL: https://doi.org/10.1147/sj.71.0015 (cit. on p. 8).

[110] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *J. ACM* 20.1 (1973), pp. 46–61. DOI: 10.1145/321738.321743. URL: http://doi.acm.org/10.1145/321738.321743 (cit. on pp. 16, 31).

[111] Robert Love. *Linux Kernel Development (2nd Edition)*. Novell Press, 2005. ISBN: 0672327201 (cit. on p. 25).

[112] Robert Love. *Linux Kernel Development (3rd Edition)*. Addison-Wesley Professional, 2010. ISBN: 0672329468 (cit. on p. 27).

[113] Jean-Pierre Lozi, Baptiste Lepers, Justin R. Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. "The Linux scheduler: a decade of wasted cores." In: *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. Ed. by Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues. ACM, 2016, 1:1–1:16. DOI: 10.1145/2901318.2901326. URL: https://doi.org/10.1145/2901318.2901326 (cit. on pp. 61, 72, 100).

[114] *Lua 5.1 Reference Manual - Coroutines*. URL: http://www.lua.org/manual/5.1/manual.html#2.11 (cit. on p. 38).

[115] Linux Programmer's Manual. *sched(7) – Linux manual page*. URL: https://man7.org/linux/man-pages/man7/sched.7.html (cit. on p. 31).

[116] Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. "Runtime support for multicore Haskell." In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. Ed. by Graham Hutton and Andrew P. Tolmach. ACM, 2009, pp. 65–78. DOI: 10.1145/1596550.1596563. URL: https://doi.org/10.1145/1596550.1596563 (cit. on p. 38).

[117] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. "Evaluation of CPU frequency transition latency." In: *Comput. Sci. Res. Dev.* 29.3-4 (2014), pp. 187–195. DOI: 10.1007/s00450-013-0240-x. URL: https://doi.org/10.1007/s00450-013-0240-x (cit. on p. 82).

[118] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. USA: Addison Wesley Longman Publishing Co., Inc., 1996. ISBN: 0201549794 (cit. on p. 34).

[119] Phoronix Media. *Phoronix Test Suite – Linux Testing & Benchmarking Platform, Automated Testing, Open-Source Benchmarking*. URL: http://www.phoronix-test-suite.com (cit. on pp. 88, 116).

[120] Andreas Merkel, Jan Stoess, and Frank Bellosa. "Resource-conscious scheduling for energy efficiency on multicore processors." In: *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*. Ed. by Christine Morin and Gilles Muller. ACM, 2010, pp. 153–166. DOI: 10.1145/1755913.1755930. URL: https://doi.org/10.1145/1755913.1755930 (cit. on pp. 16, 17).

[121] *Microsoft Docs - Fibers - Win32 apps*. URL: https://docs.microsoft.com/fr-fr/windows/win32/procthread/fibers (cit. on p. 38).

[122] *Microsoft Docs - Managing Hyper-V hypervisor scheduler types*. 2018. URL: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types (cit. on p. 40).

[123] *Microsoft Docs - Scheduling - Win32 apps*. 2018. URL: https://docs.microsoft.com/en-us/windows/win32/procthread/scheduling (cit. on p. 35).

[124] *Microsoft Docs - User-Mode Scheduling*. 2018. URL: https://docs.microsoft.com/en-us/windows/win32/procthread/user-mode-scheduling (cit. on p. 36).

[125] Ingo Molnar. *BFS vs. mainline scheduler benchmarks and measurements*. 2009. URL: https://lkml.org/lkml/2009/9/6/136 (cit. on p. 33).

[126] Hisham Muhammad. *htop*. URL: https://github.com/hishamhm/htop (cit. on p. 70).

[127] Gilles Muller, Julia L. Lawall, and Hervé Duchesne. "A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation." In: *Ninth IEEE International Symposium on High Assurance Systems Engineering (HASE 2005), 12-14 October 2005, Heidelberg, Germany*. IEEE Computer Society, 2005, pp. 56–65. DOI: 10.1109/HASE.2005.1. URL: https://doi.org/10.1109/HASE.2005.1 (cit. on p. 45).

[128] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. "CMC: A Pragmatic Approach to Model Checking Real Code." In: *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. 2002. URL: http://www.usenix.org/events/osdi02/tech/musuvathi.html (cit. on p. 61).

[129] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. "Hyperkernel: Push-Button Verification of an OS Kernel." In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 252–269. DOI: 10.1145/3132747.3132748. URL: https://doi.org/10.1145/3132747.3132748 (cit. on p. 61).

[130] *Nutanix Bible - Book of AHV*. URL: https://nutanixbible.com/#anchor-book-of-ahv (cit. on p. 40).

[131] *Oracle Docs - Using WebLogic JRockit 8.1 SDK - Selecting and Running a Thread System*. URL: https://docs.oracle.com/cd/E13188_01/jrockit/docs81/userguide/threads.html (cit. on p. 38).

[132] *Oracle VM VirtualBox*. URL: https://www.virtualbox.org/ (cit. on p. 40).

[133] *Parallels Desktop 15 for Mac*. URL: https://www.parallels.com/products/desktop/ (cit. on p. 40).

[134] Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. *Thread-sensitive scheduling for SMT processors*. Tech. rep. 2000 (cit. on p. 16).

[135] Yuvraj Patel, Leon Yang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. "Avoiding scheduler subversion using scheduler-cooperative locks." In: *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. Ed. by Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer. ACM, 2020, 9:1–9:17. DOI: 10.1145/3342195.3387521. URL: https://doi.org/10.1145/3342195.3387521 (cit. on p. 15).

[136] *PCLinuxOS*. URL: https://www.pclinuxos.com/ (cit. on p. 33).

[137] *Perf Wiki*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (cit. on p. 71).

[138] Sharon E. Perl and William E. Weihl. "Performance Assertion Checking." In: *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8,*

1993. 1993, pp. 134–145. DOI: 10.1145/168619.168630. URL: https://doi.org/10.1145/168619.168630 (cit. on p. 61).

[139] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures." In: *Commun. ACM* 17.7 (1974), pp. 412–421. DOI: 10.1145/361011.361073. URL: http://doi.acm.org/10.1145/361011.361073 (cit. on p. 39).

[140] Nikita Popov. *Cooperative multitasking using coroutines (in PHP!)* 2012. URL: https://nikic.github.io/2012/12/22/Cooperative-multitasking-using-coroutines-in-PHP.html (cit. on p. 38).

[141] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. "Energy proportionality and workload consolidation for latency-critical applications." In: *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*. Ed. by Shahram Ghandeharizadeh, Sumita Barahmand, Magdalena Balazinska, and Michael J. Freedman. ACM, 2015, pp. 342–355. DOI: 10.1145/2806777.2806848. URL: https://doi.org/10.1145/2806777.2806848 (cit. on p. 17).

[142] *ps(1) — Linux manual page*. URL: https://www.man7.org/linux/man-pages/man1/ps.1.html (cit. on p. 70).

[143] William Pugh. *Concurrent Maintenance of Skip Lists*. Tech. rep. USA, 1990 (cit. on p. 33).

[144] Qualcomm. *MSM8X60/APQ8060 – Snapdragon™ Dual-Core Mobile Processor*. 2011. URL: https://www.qualcomm.com/media/documents/files/snapdragon-msm8x60-apq8060-product-brief.pdf (cit. on p. 80).

[145] *Quick Reference Guide for Intel® Core™ Processor Graphics*. URL: https://software.intel.com/content/www/us/en/develop/articles/quick-reference-guide-to-intel-processor-graphics.html (cit. on p. 8).

[146] Milos Rancic. *The World's First Computer Operating System Implemented at General Motors Research Labs in Warren, Michigan in 1955*. 2007. URL: https://millosh.wordpress.com/2007/09/07/the-worlds-first-computer-operating-system-implemented-at-general-motors-research-labs-in-warren-michigan-in-1955/ (cit. on p. 11).

[147] Melissa A. Rau and Evgenia Smirni. "Adaptive CPU Scheduling Policies for Mixed Multimedia and Best-Effort Workloads." In: *MASCOTS 1999, Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 24-28 October, 1999, College Park, Maryland, USA*. IEEE Computer Society, 1999, pp. 252–261.

DOI: 10.1109/MASCOT.1999.805062. URL: https://doi.org/10.1109/MASCOT.1999.805062 (cit. on p. 16).

[148] Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013. ISBN: 978-3-642-32026-2. DOI: 10.1007/978-3-642-32027-9. URL: https://doi.org/10.1007/978-3-642-32027-9 (cit. on p. 15).

[149] Jeff Roberson. "ULE: A Modern Scheduler for FreeBSD." In: *Proceedings of BSDCon 2003, San Mateo, California, USA, September 8-12, 2003*. Ed. by Gregory Neil Shapiro. USENIX, 2003, pp. 17–28. URL: http://www.usenix.org/publications/library/proceedings/bsdcon03/tech/roberson.html (cit. on p. 34).

[150] Michael Roitzsch, Stefan Wachtler, and Hermann Härtig. "ATLAS: Look-ahead scheduling using workload metrics." In: *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. IEEE Computer Society, 2013, pp. 1–10. DOI: 10.1109/RTAS.2013.6531074. URL: https://doi.org/10.1109/RTAS.2013.6531074 (cit. on p. 16).

[151] Steven Rostedt. *KernelShark*. URL: https://kernelshark.org/ (cit. on p. 71).

[152] Steven Rostedt. *Linux Kernel Documentation - ftrace - Function Tracer*. URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt (cit. on p. 64).

[153] Steven Rostedt. *trace-cmd: utilities for Linux ftrace*. URL: https://github.com/rostedt/trace-cmd (cit. on p. 70).

[154] Saowanee Saewong, Ragunathan Rajkumar, John P. Lehoczky, and Mark H. Klein. "Analysis of Hierar hical Fixed-Priority Scheduling." In: *14th Euromicro Conference on Real-Time Systems (ECRTS 2002), 19-21 June 2002, Vienna, Austria, Proceedings*. IEEE Computer Society, 2002, pp. 173–181. DOI: 10.1109/EMRTS.2002.1019197. URL: https://doi.org/10.1109/EMRTS.2002.1019197 (cit. on p. 15).

[155] Dongyou Seo, Hyeonsang Eom, and Heon Y. Yeom. "MLB: A Memory-aware Load Balancing for Mitigating Memory Contention." In: *2014 Conference on Timely Results in Operating Systems, TRIOS '14, Broomfield, CO, USA, October 5, 2014*. Ed. by Ken Birman. USENIX Association, 2014. URL: https://www.usenix.org/conference/trios14/technical-sessions/presentation/seo (cit. on p. 16).

[156] Gene Shekhtman and Mike Abbott. *State Threads Library for Internet Applications*. URL: http://state-threads.sourceforge.net/ (cit. on p. 39).

[157] Kai Shen, Ming Zhong, and Chuanpeng Li. "I/O System Performance Debugging Using Model-driven Anomaly Characterization." In: *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*. 2005. URL: http://www.usenix.org/events/fast05/tech/shen.html (cit. on p. 61).

[158] Hyotaek Shim and Sung-Min Lee. *CFS-v: I/O Demand-driven VM Scheduler in KVM*. 2014. URL: https://www.linux-kvm.org/images/e/ee/03x06-CFS-v.pdf (cit. on p. 40).

[159] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. "Push-Button Verification of File Systems via Crash Refinement." In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 2016, pp. 1–16. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson (cit. on p. 61).

[160] Mark K. Smotherman, Edward H. Sussenguth Jr., and Russell J. Robelen. "The IBM ACS Project." In: *IEEE Annals of the History of Computing* 38.1 (2016), pp. 60–74. DOI: 10.1109/MAHC.2015.50. URL: https://doi.org/10.1109/MAHC.2015.50 (cit. on p. 7).

[161] Allan Snavely and Dean M. Tullsen. "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor." In: *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000*. Ed. by Larry Rudolph and Anoop Gupta. ACM Press, 2000, pp. 234–244. DOI: 10.1145/356989.357011. URL: https://doi.org/10.1145/356989.357011 (cit. on p. 16).

[162] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. "Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems." In: *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*. Ed. by Shan Lu and Erik Riedel. USENIX Association, 2015, pp. 529–540. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/srikanthan (cit. on p. 16).

[163] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. "Coherence Stalls or Latency Tolerance: Informed CPU Scheduling for Socket and Core Sharing." In: *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. Ed. by Ajay Gulati and Hakim Weatherspoon. USENIX Association, 2016, pp. 323–336. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/srikanthan (cit. on p. 16).

[164] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. "A Feedback-driven Proportion Allocator for Real-Rate Scheduling." In: *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*. Ed. by Margo I. Seltzer and Paul J. Leach. USENIX Association, 1999, pp. 145–158. URL: https://dl.acm.org/citation.cfm?id=296820 (cit. on p. 16).

[165] Honglin Su. *Oracle VM 2.2 New Feature: CPU Scheduling Priority and Cap*. 2009. URL: https://blogs.oracle.com/virtualization/oracle-vm-22-new-feature:-cpu-scheduling-priority-and-cap (cit. on p. 40).

[166] Minyoung Sung, Soyoung Kim, Sangsoo Park, Naehyuck Chang, and Heonshik Shin. "Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread." In: *Inf. Process. Lett.* 84.4 (2002), pp. 221–225. DOI: 10.1016/S0020-0190(02)00286-7. URL: https://doi.org/10.1016/S0020-0190(02)00286-7 (cit. on p. 37).

[167] *Tcl8.6.10/Tk8.6.10 Documentation - Tcl Commands - coroutine*. URL: http://www.tcl.tk/man/tcl8.6/TclCmd/coroutine.htm (cit. on p. 38).

[168] *The CPU Scheduler in VMware vSphere 5.1*. Tech. rep. VMware. URL: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-vsphere-cpu-sched-performance-white-paper.pdf (cit. on p. 40).

[169] *The Kaffe Virtual Machine*. URL: https://github.com/kaffe/kaffe (cit. on p. 38).

[170] *The Python Standard Library - Coroutines and Tasks*. URL: https://docs.python.org/3.7/library/asyncio-task.html (cit. on p. 38).

[171] *The Racket Reference*. URL: https://docs.racket-lang.org/reference/eval-model.html#%28part._thread-model%29 (cit. on p. 38).

[172] *The Stackless Python programming language*. URL: https://github.com/stackless-dev/stackless/wiki (cit. on p. 39).

[173] Linus Torvalds. *Re: Just a second ...* 2001. URL: https://lore.kernel.org/lkml/Pine.LNX.4.33.0112151603180.4493-100000@penguin.transmeta.com/ (cit. on p. 24).

[174] Linus Torvalds. *Re: [ANNOUNCE] RSDL completely fair starvation free interactive cpu scheduler*. 2007. URL: https://lore.kernel.org/lkml/Pine.LNX.4.64.0703082226530.10832@woody.linux-foundation.org/ (cit. on p. 68).

[175] Manohar Vanga, Arpan Gujarati, and Björn B. Brandenburg. "Tableau: a high-throughput and predictable VM scheduler for high-density workloads." In: *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. Ed. by Rui Oliveira, Pascal Felber, and Y. Charlie Hu. ACM, 2018, 28:1–28:16. DOI: 10.1145/3190508.3190557. URL: https://doi.org/10.1145/3190508.3190557 (cit. on p. 40).

[176] *Virtual-machine scheduling and scheduling in virtual machines*. 2019. URL: https://lwn.net/Articles/793375/ (cit. on p. 40).

[177] Dmitry Vyukov. *Scalable Go Scheduler Design Doc*. 2012. URL: https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit?usp=sharing (cit. on p. 38).

[178] Carl A. Waldspurger and William E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management." In: *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, California, USA, November 14-17, 1994*. USENIX Association, 1994, pp. 1–11. URL: http://dl.acm.org/citation.cfm?id=1267639 (cit. on p. 15).

[179] *Welcome to bhyve - The BSD Hypervisor*. URL: https://bhyve.org/ (cit. on p. 40).

[180] WikiChip. *Thermal Velocity Boost (TVB) - Intel*. URL: https://en.wikichip.org/wiki/intel/thermal_velocity_boost (cit. on p. 79).

[181] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. "The worst-case execution-time problem - overview of methods and survey of tools." In: *ACM Trans. Embedded Comput. Syst.* 7.3 (2008), 36:1–36:53. DOI: 10.1145/1347375.1347389. URL: https://doi.org/10.1145/1347375.1347389 (cit. on p. 32).

[182] Maurice V. Wilkes. "Slave Memories and Dynamic Storage Allocation." In: *IEEE Trans. Electronic Computers* 14.2 (1965), pp. 270–271. DOI: 10.1109/PGEC.1965.264263. URL: https://doi.org/10.1109/PGEC.1965.264263 (cit. on p. 8).

[183] Rafael J. Wysocki. *CPU Performance Scaling*. 2017. URL: https://www.kernel.org/doc/html/v5.4/admin-guide/pm/cpufreq.html (cit. on p. 80).

[184] Rafael J. Wysocki. *intel_pstate CPU Performance Scaling Driver*. 2017. URL: https://www.kernel.org/doc/html/v5.4/admin-guide/pm/intel_pstate.html (cit. on p. 81).

[185] *Xen Wiki - Credit Scheduler*. URL: https://wiki.xenproject.org/wiki/Credit_Scheduler (cit. on p. 40).

[186] *Xen Wiki - Credit2 Scheduler*. URL: https://wiki.xenproject.org/wiki/Credit2_Scheduler (cit. on p. 40).

[187] *Xen Wiki - RTDS-Based-Scheduler*. URL: https://wiki.xenproject.org/wiki/RTDS-Based-Scheduler (cit. on p. 40).

[188] Di Xu, Chenggang Wu, and Pen-Chung Yew. "On mitigating memory bandwidth contention through bandwidth-aware scheduling." In: *19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010*. Ed. by Valentina Salapura, Michael Gschwind, and Jens Knoop. ACM, 2010, pp. 237–248. DOI: 10.1145/1854273.1854306. URL: https://doi.org/10.1145/1854273.1854306 (cit. on p. 16).

[189] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. "Using Model Checking to Find Serious File System Errors (Awarded Best Paper!)" In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. 2004, pp. 273–288. URL: http://www.usenix.org/events/osdi04/tech/yang.html (cit. on p. 61).

[190] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. "Redline: First Class Support for Interactivity in Commodity Operating Systems." In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 73–86. URL: http://www.usenix.org/events/osdi08/tech/full_papers/yang/yang.pdf (cit. on p. 16).

[191] *Zenwalk GNU Linux*. URL: http://zenwalkgnulinux.blogspot.com/ (cit. on p. 33).

[192] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. "CPI^2: CPU performance isolation for shared compute clusters." In: *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*. Ed. by Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek. ACM, 2013, pp. 379–391. DOI: 10.1145/2465351.2465388. URL: https://doi.org/10.1145/2465351.2465388 (cit. on p. 16).

[193] Ming Zhao and Jorge Cabrera. "RTVirt: enabling time-sensitive computing on virtualized systems through cross-layer CPU scheduling." In: *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. Ed. by Rui Oliveira, Pascal Felber, and Y. Charlie Hu. ACM, 2018, 27:1–27:13. DOI: 10.1145/3190508.3190527. URL: https://doi.org/10.1145/3190508.3190527 (cit. on p. 40).

[194] Alin Zhong, Hai Jin, Song Wu, Xuanhua Shi, and Wei Gen. "Optimizing Xen Hypervisor by Using Lock-Aware Scheduling." In: *2012 Second International Conference on Cloud and Green Computing, CGC 2012, Xiangtan, Hunan, China, November 1-3, 2012*. Ed. by Jianxun Liu, Jinjun Chen, and Guandong Xu. IEEE Computer Society, 2012, pp. 31–38. DOI: 10.1109/CGC.2012.115. URL: https://doi.org/10.1109/CGC.2012.115 (cit. on p. 15).

[195] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing shared resource contention in multicore processors via scheduling." In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*. Ed. by James C. Hoe and Vikram S. Adve. ACM, 2010, pp. 129–142. DOI: 10.1145/1736020.1736036. URL: https://doi.org/10.1145/1736020.1736036 (cit. on p. 16).

INDEX

ACRONYMS

AGU address generation unit. 7
ALU arithmetic logic unit. 7
API application programming interface. 16, 22, 23, 38, 45, 54, 56, 68, 72, 115, 118, 128, 131, 132

BFS Brain Fuck Scheduler. 32, 33, 69
BKL Big Kernel Lock. 21
BSD Berkeley Software Distribution. 34

CBS Constant Bandwidth Server. 31
CFS Completely Fair Scheduler. 3, 24, 27–35, 40, 41, 43, 46, 50, 52, 55, 59, 62–65, 67, 69, 70, 74–78, 81, 86–100, 102, 103, 106–114, 116–123, 126, 128, 129, 131–133
CMP chip-level multiprocessing. 7–9
CPU central processing unit. 1, 3, 4, 11, 13, 15, 18–21, 23, 24, 27–32, 34, 36, 40, 43, 55, 70, 72, 73, 79–84, 86–89, 93, 94, 98, 101, 103, 104, 112, 116, 129, 131–133

DRAM dynamic random access memory. 88
DSL domain-specific language. 2, 4, 41, 43–48, 50, 57, 59–61, 65, 67–69, 104, 105, 131, 132
DVFS dynamic voltage and frequency scaling. 8, 17, 79, 80, 132, 137

EDF Earliest Deadline First. 16, 31
EIST Enhanced Intel SpeedStep Technology. 79, 82

FIFO First-In First-Out. 31, 51, 52, 67
FPU floating-point unit. 7
FTL frequency transition latency. 79, 82–86, 104, 132, 133

GHC Glasgow Haskell Compiler. 38

HPC High Performance Computing. 38, 132

I/O input/output. 1, 9, 11–13, 19–21, 24, 34, 37, 40, 46, 47, 62, 63, 71, 114, 129
IP instruction pointer. 5, 7

LLC last level cache. 29

MSR model-specific register. 82, 83
MuQSS Multiple Queue Skiplist Scheduler. 32, 33

NUMA non-uniform memory access. 1, 9, 10, 15, 29, 31, 36, 48, 64, 71, 87, 88, 98, 102, 109, 114

OPP Operating Performance Point. 80
OS operating system. 1–6, 9, 11–15, 17–19, 24, 25, 34–41, 43, 61, 62, 67, 79, 80, 82, 105, 110, 111, 118, 131, 132

PID process identifier. 22, 58, 59, 73
POSIX Portable Operating System Interface. 18, 19, 23, 31
PPID parent process identifier. 22

RAPL running average power limit. 88, 94
RCU read-copy-update. 55

SMP symmetric multiprocessing. 9, 15, 21, 25, 26, 34, 35, 37, 109
SMT simultaneous multithreading. 7–9, 14, 16, 29, 37, 48, 83, 100, 115, 118
SP stack pointer. 5, 7

UMS User-Mode Scheduling. 36, 38

vCPU virtual CPU. 18, 40, 46, 48
VM virtual machine. 18, 37, 38, 40, 48

WCET Worst-Case Execution Time. 32