OS-level frameworks for rapid prototyping of process scheduling algorithms on Linux

Final Project for the Degree in Computer Science and Engineering

Carlos Bilbao Muñoz

Advisor: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática (grupo bilingüe)

Facultad de Informática

Universidad Complutense de Madrid

Madrid, Junio de 2020

OS-level frameworks for rapid prototyping of process scheduling algorithms on Linux

Final Project Report

Carlos Bilbao Muñoz

Advisor: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática (Grupo Bilingüe)

Facultad de Informática

Universidad Complutense de Madrid

Madrid, Junio de 2020

"No one told you when to run, you missed the starting gun." - Pink Floyd

Acknowledgments

I am grateful to my advisor, Juan Carlos Sáez Alcaide, from whom I feel very fortunate to have been able to learn. Thanks to him I was not only able to get my first internship, but also meet great people and discovered the fun of research along the way.

I also want to thank my parents for their unconditional support.

Finally, for all the nice memories I would like to thank the friends I have made these years at UCM.

Abstract

Linux is equipped with multiple scheduling algorithms that are implemented as separate scheduling classes. Unfortunately, due to the current design of the scheduler these classes have to be created inside the kernel, which makes difficult to develop and debug new algorithms. In particular, for each new bug detected in the implementation of the algorithm it is necessary to recompile the kernel and reboot the system, thus making development at this level a time-consuming process.

Some researchers resorted to prototyping scheduling algorithms via *user space* programs that use system calls to drive thread-to-core assignments. Nonetheless, this approach comes at the expense of an important overhead and many relevant low-level events cannot simply be monitored outside the kernel. At the same time, recent research has proven that the system performance can be severely degraded as a consequence of the simultaneous execution of threads that make intensive use of memory resources. Hence, the Linux scheduler has substantial room for improvement.

Based on these observations, we established two objectives for this project: easing the prototyping of new scheduling algorithms, and putting into practice several solutions proposed for the shared-resource contention problem but that were never implemented at the kernel level.

First, we strived to address the prototyping problem with a scheduling framework that could perform dynamic insertion of scheduling algorithms at runtime, without rebooting the system. We tackled this with a new variant of the Linux kernel, the FSAC, *Framework for Scheduling Algorithm Creation* kernel. Subsequently, we included support for prototyping in the open-source monitoring tool PMCTrack, and named this **PMCSched**. We did so to be able to develop resource-conscious scheduling algorithms, since PMCTrack already had all the support required for monitoring memory-related events such as cache-misses or memory bandwidth consumption. This second framework, focused on co-scheduling and monitoring, was intended for scheduling algorithms with different requirements. When the PMCSched framework was completed we developed a contention-aware scheduling algorithm *MEMSCHED* with which we intended to solve the aforementioned shared-resource contention problem. We also performed a comprehensive experimental evaluation of the implemented scheduling algorithm using a system equipped with an octa-core Intel processor, and compared its effectiveness with that of the current Linux scheduler.

Keywords: Scheduling, Linux kernel, multicore processors, shared-resource contention

Resumen

Linux está equipado con múltiples algoritmos de planificación implementados como separadas clases de planificación. Desafortunadamente, debido al actual diseño del planificador estas clases deben ser creadas dentro del propio kernel, lo que dificulta el desarrollo y depuración de nuevos algoritmos. Esencialmente, por cada nuevo bug detectado en la implementación del algoritmo, es necesario recompilar el kernel y reiniciar el sistema, haciendo que el desarrollo a este nivel consuma mucho tiempo.

Algunos investigadores intentaron prototipar algoritmos de planificación desde programas en *espacio de usuario*, empleando llamadas al sistema para forzar las asignaciones hilo-a-core. Sin embargo, esta estrategia es a expensas de una importante sobrecarga y muchos eventos de bajo nivel no pueden ser simplemente monitorizados fuera del kernel. Al mismo tiempo, investigación reciente ha demostrado que el rendimiento del sistema se puede ver severamente degradado como consecuencia de la ejecución simultánea de hilos que hacen uso intensivo de los recursos de memoria. Por tanto, el planificador de Linux tiene gran espacio para mejoras.

Basados en estas observaciones, establecimos dos objetivos para este proyecto: facilitar el prototipado de nuevos algoritmos de planificación y poner en práctica varias soluciones propuestas para el problema de contención de recursos, que nunca habían sido puestas en práctica al nivel del kernel.

Primero, para afrontar el problema del prototipado ideamos un framework de planificado, que podía realizar inserciones dinámicas en tiempo de ejecución, sin reiniciar el sistema. Esto supuso la creación de una variante del kernel de Linux, FSAC *Framework for Scheduling Algorithm Creation*. Seguidamente, incluimos todo el soporte necesario para prototipado en la herramienta de monitorización open-source PMCTrack, y llamamos a esta nueva utilidad **PMCSched**. Hicimos esto para ser capaces de desarrollar algoritmos conscientes de la situación de los recursos, dado que PMCTrack ya contaba con todo el soporte requerido para la monitorización de eventos relativos a memoria tales como los fallos de cache o el consumo de ancho de banda. Este segundo framework, centrado en el planificado grupal, fue diseñado para algoritmos con diferentes requisitos. Cuando el framework PMCSched fue completado, desarrollamos un algoritmo consciente de la contención *MEMSCHED* con el cual pretendíamos solucionar el previamente citado problema de contención de recursos. Realizamos así mismo una exhaustiva evaluación experimental del algoritmo de planificación implementado empleando un sistema octa-core Intel, y comparamos su eficiencia con la del del actual planificador de Linux.

Palabras clave: Planificado, kernel Linux, procesadores multicore, contención de recursos compartidos

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The justification for all the work carried out in this Final Project requires of an analysis of current market trends, and background regarding two of the most important technical challenges that are to be faced in the near future: the prototyping of new scheduling algorithms and the shared resource contention problem. Combined they prove the Linux kernel will gain a decisive role in the computer science landscape in the next few years. In this chapter, we will support this claim and provide an overview of the project, which is closely related to the aforementioned tendencies and current scientific efforts. Similarly, we will explain the work plan. We will conclude with a comprehensive summary of the contents of this report.

The entire project is divided in two parts, and this report has been structured accordingly. Two major technical challenges are going to be identified below, followed by an explanation of how we decided to address them. Both of them are closely related to the research on the Systems area. That being said, it is reasonable to state that they show potential implications in others.

1.1 Motivation

1.1.1 The Memory Wall and Power Wall problems

One of the major challenges in computer architecture is the so-called **Power Wall**. The physical limits of the CMOS technology are about to be reached, and energy efficiency needs to be taken into account more than ever. Moreover, the technical difficulties associated with the production at the current scale increase manufacturing costs. At this point, only three companies are still trying to fulfill the Moore's prophecy (Intel, Samsung Electronics and Taiwan Semiconductor Manufacturing Company Limited or TSMC), even-though they are facing an undeniable decrease in economic profitability along with a decline in the integration capabilities, as shown in Figure 1.1. As a result, some researchers have begun to refer to this period as the **"Post-Moore Era"** [1].

New tendencies have emerged in the market due to these well-known constraints. Nowadays, some companies try to integrate an increasingly number of cores per chip -while maintaining

moderate processor frequencies- and others combine cores and/or accelerators on the same platform leading to a wide spectrum of **heterogeneous architectures** [2].

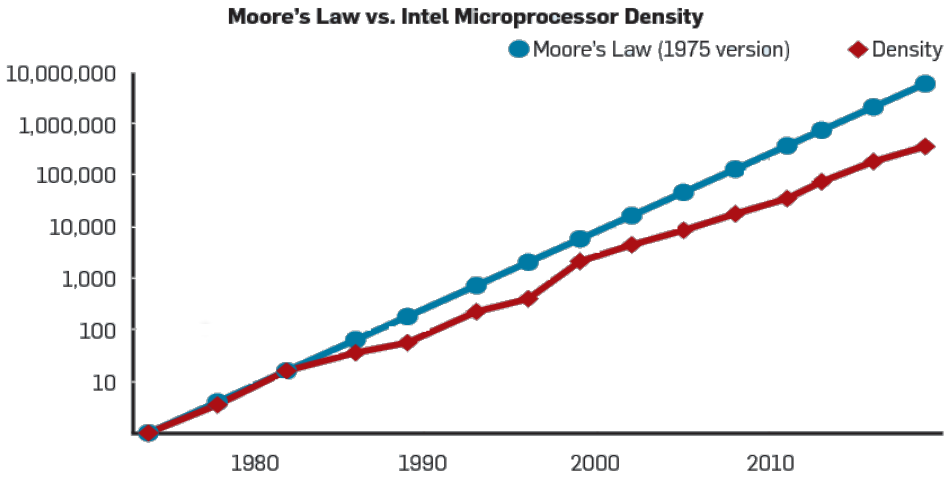


Figure 1.1: Number of transistors per chip on Intel processors compared to the prediction made by Gordon Moore in 1975 [1]

Nevertheless, both strategies pose important challenges. In the first approach (i.e. symmetric multicore architectures) the increasing number of cores per chip aggravates the problem of shared resource contention. In particular, all of the processes running simultaneously may compete for shared memory resources [3], such as the DRAM controller, the last-level cache (LLC), and the bus or interconnection network. As a consequence, shared resource contention aggravates the **Memory Wall** problem. Heterogeneous architectures appear to be more promising when it comes to memory efficiency. Nevertheless, they also show important limitations related to programming flexibility since one needs to cope with different Instruction Set Architectures (ISAs). One alternative to deal with this are the **Asymmetric Multicore Processors** (AMPs) that combine complex big cores with power efficient ones that have less energetic cost associated. These are widely extended in mobile devices. Take for instance, the ARM big.LITTLE processor family [4].

In sum, there is no doubt that the market shows a tendency to specialization. Heterogeneity has proven to be the most viable solution and there is a growing demand to meet in the academic world focused on Systems research. The main efforts of this project were oriented towards adapting GNU/Linux and in particular its kernel to this emerging trend. Broadly speaking, the main goal was to ease the job of OS (operating systems) developers dealing with memory-related challenges on multicore architectures.

1.1.2 The challenges of OS development

The GNU/Linux operating system is used by millions in a wide variety of distributions. It can be found in all forms and shapes, from versions focused on security (Kali Linux, Parrot,

BlackArch,…) to extremely lightweight systems that fit almost on any platform (Linux Lite, Ubuntu Mate,…). Google invested a king's ransom on Chromebook, and Android -which is based on the Linux kernel- continues to be the absolute leader in mobile telephony worldwide.

The main thing all these distributions and products have in common is the Linux kernel. The *kernel* is to the operating system (OS) what the CPU is to the hardware. The OS simply cannot be understood without the kernel, the core and central decision-making component. Every important choice regarding the performance, each of the system calls and all the internal file handling, the crucial memory management, etc. Virtually anything relevant ends up in the kernel and goes all the way back to the end user. Furthermore, alone with the firmware, it is the closest software to "the metal". Consequently, there is not much room for abstractions, and a deep understanding of the hardware itself is required for the most part. This becomes particularly true when dealing with drivers, processors or network connections. It is such a complex and broad topic, it is hard not to appreciate its brilliance as an engineer. Notably, the Linux kernel has become so prevalent in both OS research and development that it may be considered the norm [2], [5]–[7]. This standardization is not by chance, for it is a fairly modular, thought and carefully reviewed system. Besides, the lessons learned can be easily applied to other systems such as the Android kernel or the XNU Apple kernel. The latter is derived from FreeBSD, which was strongly based upon UNIX, just like Linux. Similarly, the Android kernel is a reshape of Linus Torvalds' creation with some years of development by Google -and the company they acquired- in a different direction.

Granted, being the Linux kernel such an important component of the OS and subsequently to the performance of the system as a whole, it is no surprise that a significant amount of related work and research has been conducted over the years. Almost every line of code has been rigorously reviewed by skilled experts and many new features along with fixes are periodically included for each new version released. There is, nonetheless, big room for improvement. Every time the processor is upgraded, new ways to take advantage of its features are made available in the kernel. In this way, both disciplines keep boosting each other. For example, Intel recently launched processors with technologies that support last-level cache LLC monitoring, all collected in the Intel RDT (Intel® Resource Director Technology) [8]–[12] that allow kernel developers to distribute shared resources among cores more fruitfully now.

A lot of money and brain-power is invested every year to improve the CPU efficiency. Nevertheless, like in any other system or situation, **how tasks are executed is as important as what tasks are those.** When finding ways to improve efficiency and optimizing production, order matters. When dealing with several processes sharing one same resource (i.e. the CPU or main memory), different strategies known as scheduling algorithms are used to discern what task to execute next, or at a given time. The scheduling algorithms are included in the kernel as part of the process *scheduler*. A more detailed description of the scheduler will be provided in the next chapter.

Many **scheduling algorithms** have been proposed over the years. In the last decade, the scientific community has witnessed great progress in the design and development of such algorithms [3], [6]. The main goal pursued was fairness [2], [3], [6] delivered by efficient and modular designs, or even guaranteeing some real-time constraints for critical systems. Nevertheless, even with such great work carried out theoretically, **the development of**

systems that actually implement those algorithms remains a challenge. This is due to the difficulties that arise with the the actual implementation, the hands-on work, that suffers from some inherent limitations. The *preemptive* SMP Linux kernel is a complex piece of software that requires a high level of understanding of its workflow and structures, even more when coping with several processors and hyper-threading, interrupts or lacking C libraries. Instead of exposing a subset of the ISA, all the instructions are available. Similarly, the amount of usable registers is larger than at *user space*. Developers are sometimes forced to use assembly instructions directly, for very specific tasks that are architecture dependent. Debugging the Linux kernel is a very demanding exercise of patience, and recompiling it for every change can be extenuating [13]. Many times a new kernel is compiled and booted only to find a *kernel panic* (the Windows equivalent is the "*Blue Screen of Death*") at the end. Needless to say, few or no files with relevant information of the error encountered can be found after rebooting in a safe manner. Some parts of the Linux kernel, such as the process scheduler, were simply not conceived to be modified much, which makes the job of researchers more complicated.

1.2 Project goals

All of this leads to the main objectives of this final project, related to the market trends and the two technical challenges described in the previous subsections. Two major unsolved problems that could be addressed in the systems area have been explained:

1. The design, debugging and **prototyping of new scheduling algorithms** is an enormous challenge even for experienced system developers. To this day, this was not done dynamically (i.e. at runtime without recompiling the kernel), thus complicating the process even more. The limitations of techniques focused on user-space solutions and their constraints accessing hardware monitoring facilities have greatly staggered the development of new scheduling algorithms and made kernel-level frameworks a very likely future to harness advances in OS development [6].

2. Current architectures pose a new challenge. Academic research in state-of-the-art multicore systems proved that there is non-negligible performance degradation linked to shared resource contention [14]. This is because threads may compete for memory resources, not only cache but also memory bandwidth, an aspect that current kernel schedulers do not fully take into account [15], [16].

Accordingly, **this project diverged in two directions** that aimed to address both issues:

1. First, a variant of the Linux kernel that helps dealing with the general design and prototyping of scheduling algorithms, FSAC (*Framework for Scheduling Algorithms Creation*) was developed. This framework avoids hours of kernel recompilation and allows the dynamic patching of the scheduler without rebooting the system, as it can be done at runtime. Its development was based on a previous project LITMUS-RT [5], which did not allow dynamic prototyping but had some useful abstractions already implemented.

This project is still in a development stage -and will most likely remain so for some time- but the main abstractions and foundations have already been provided. Specifically, thousands of lines of kernel code have been written, nineteen files have been modified and fourteen new others created.

2. Secondly, other framework was developed inside the open-source PMCTrack tool [7], to allow the prototyping of resource-conscious scheduling algorithms. We named it **PMCSched**. PMCTrack already had all the support required for monitoring memory-related events such as cache-misses or memory bandwidth consumption, leveraging Intel advanced monitoring and hardware allocation technologies, as well as access to other hardware monitoring information (such as energy consumption). This patch constituted nearly three thousand lines of code.

Summing them and patches to related projects, we can conclude this project involved writing more than **six thousand lines of kernel code**. It is worth noting that while FSAC is a novel kernel version for basic general scheduling, PMCSched is a patch that contributes to a refined open-source effort. Thanks to all the prior work on PMCTrack, we were able to finish the second infrastructure. The new version offers support for resource-conscious scheduling algorithms using cache and bus bandwidth hardware monitoring technologies.

PMCSched and the FSAC kernel are both OS-level frameworks for prototyping scheduling algorithms, but **they have very different implementations, advantages and purpose**. While FSAC offers a very fine-grained control for efficient scheduler implementations -at the expense of a higher development effort- PMCSched is optimized for co-scheduling and for leveraging information from performance monitoring counters. FSAC offers absolute control over the core's run-queues and flexibility, but PMCSched can take advantage of other utilities included on PMCTrack, such as the scheduling of selected tasks or the system information collected at runtime.

1.3 Work Plan

Regarding the work plan of this project, it was based on dividing efforts between both goals. We had to think through what we intended to achieve and break the project into smaller milestones, the progress of which could then be monitored with some collaboration tool such as Trello. The main stages that were set to be achieved are the following:

- Some time had to be spend studying prior work. It was decided that this should last no more than three months, ending around December 2019. This meant reviewing the LITMUS-RT project, its publications and articles, along with extensive documentation on the Linux kernel.
- After that, we had to be able to get down to the code. Regarding FSAC, we decided to start with the headers, which made another month of work. We decided that by the end of March 2020 we had to have the infrastructure finished.
- Meanwhile, we had to start developing the API for the PMCSched plugins. We planned to start implementing the FSAC scheduling class as early as the second semester, while

writing some test algorithm for PMCSched that could be finished for testing by April 2020. Luckily, we not only managed to implement a coarse-grained Round Robin (RR) algorithm along with a memory-cognizant solution [17] using Intel technologies -as planned- but also finished a group co-scheduling algorithm.

- At a minimum, another two months (April and May 2020) had to be left available to correct potential bugs that could be found in both the FSAC implementation and the new PMCSched tool.
- Finally, the remaining time had to be invested in carrying out experiments with PMCSched, publishing FSAC code on GitHub and writing this report.

We were able to follow this work plan to the letter, except for minor adjustments resulting from the Covid-19 situation. During the experimental stage, a script that generates automatic reports in .prv format [18] for the scheduler behavior was written using SystemTap. Those files were then used to create graphics using software from the Barcelona Supercomputing Center.

1.4 Structure of the Report

To conclude this chapter, it would be useful to recapitulate by guiding the reader and explaining the way the contents of this report are structured. The remainder of the report is organized in the following chapters:

- Chapter 2 *An overview of the Linux kernel scheduler*: A short review of both the Linux kernel and its scheduler is to be found here.
- Chapter 3 *Prior work and LITMUS-RT patches*: It includes prior work related to what was aimed to achieve, along with contributions to a related academic project.
- Chapter 4 *Development of the FSAC kernel*: The building of an API for the FSAC kernel is outlined. The internal Linux kernel modifications required are explained, with emphasis on the real-time support.
- Chapter 5 *PMCSched: A PMCTrack-based scheduling framework*: Development of an API for the PMCTrack Scheduling Framework. In addition, the development of the the framework itself is depicted here.
- Chapter 6 *Contention-aware scheduling algorithm*: An explanation on how to build a n-cores co-scheduling algorithm [19] and a contention-aware one [17], with PMCSched can be found here.
- Chapter 7 *Experimental Evaluation*: This chapter is devoted to the experiments with the new algorithms and the PMCSched Framework, discussing results.
- Chapter 8 *Conclusions and Future Work*.
- Appendix A *Installation and replication guide*: This section is devoted to discuss the work environment and methodology, as a way for the reader to start digging into this project and making replication possible.

- Translation into Spanish of several parts of the report including Chapters 1 and 8.

Chapter 2

An overview of the Linux kernel scheduler

2.1 The Linux kernel and the scheduler

As explained in the introductory chapter, the *kernel* is the core and main decision-making component of the operating system (OS). All GNU/Linux distributions have at least one thing in common, and that is the Linux kernel. Countless systems such as Android are greatly based on Linux, while others share ancestors and thereby inherit many common characteristics. Such is the case of the XNU Apple kernel (based on UNIX too).

In this section the Linux kernel scheduler is described, being the main software this project had to deal with. One of the first challenges faced was understanding in detail how the Linux scheduler works. As with almost everything in the Linux kernel, it was not something comprehensible at first glance, but required a moment of closer attention and the reviewing of extensive documentation. Because of this, and since it is such a specific topic that -even being a kernel developer- the reader might not be familiar with, we considered relevant devoting this section of the report to a brief but broad explanation on how does it work. This way, we intend to help the reader recognize what was fulfilled throughout this project.

The process scheduler is a critical component of any OS for it is in charge of picking the task that the CPU should execute next. The scheduler needs to be as efficient as possible, as well as fair to every waiting thread as the design can get. Results in different experimental implementations of schedulers in studies [2] (p.1706) have shown that it can indeed make a huge difference in terms of performance. The perfect scheduler should be extremely fair, decrease the latency, increase the CPU usage and manage different priority levels wisely.

In particular, the Linux scheduler is capable of managing up to 140 scheduling priority levels, being the first 100 reserved for real-time processes. The real-time tasks are those who guarantee termination within specified time constraints (budgets), something particularly important in critical systems. These priority levels are then converted to weights using a 40-value array (`prio_to_weight`) to determine -among other things- CPU time utilization.

The usage of the `nice` user command was conceived to modify that level of priority.

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Nonetheless, it is important to keep in mind that these weights are related to the CPU time each thread *could* consume. Even though it is in a sense closely related to the scheduling priority, priorities are affected by different factors that will be described later. It can be confusing at times, but it is important to differentiate. Take for instance, two equal tasks T1 and T2, and let us assume same priority conditions. Consider now that both have a priority level of `nice` = 0, which is equivalent to 1024 in load weight. The CPU usage for each of them is then the ratio between that load and the sum of the other tasks weight (1024/2048), a 50% of the CPU milliseconds. But if we decrease T2 priority level using `nice++`, then T2 weight will be mapped to 820, resulting in a CPU time of (820/1884), approximately a 45% compared to T1, with 55%.

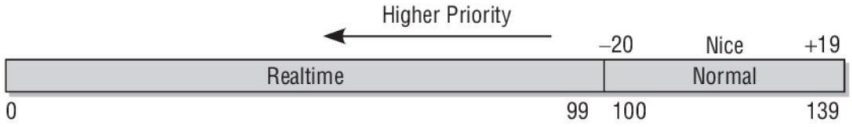


Figure 2.1: Priority levels [20]

Hence, the increase of one in the level of priority resulted in a 10% less usage of CPU time for T2 compared to T1. Anyhow, this behaviour means that **"nicing" a process by one has wildly different effects depending on the starting `nice` values**, and hence we might encounter extremely unbalanced situations [13] (p.48). Regular processes inherit by default the `nice` level of their parent, but many useful system calls are implemented to modify those aspects as desired, such as `setpriority()`, `sched_set_param()` or `sched_yield()`.

Later on these loads are very useful to another important component in multicore or multiprocessor machines, the ***Load Balancer***, which guarantees a similar weight load per run-queue. We understand "CPU weight load" as the sum of weights of the processes for a given CPU. The Load Balancer performs topology-aware thread migrations taking into account the affinity mask, a bit mask indicating in what processor the thread or process is allowed to run on. Nevertheless, this leads to associated problems, since context switches are usually computationally expensive. Furthermore, the migration of threads between run-queues block them momentarily, for they have to be manipulated acquiring a spin-lock. In addition, this requires rebuilding the cache. This is why the Linux kernel tries to avoid such scenarios.

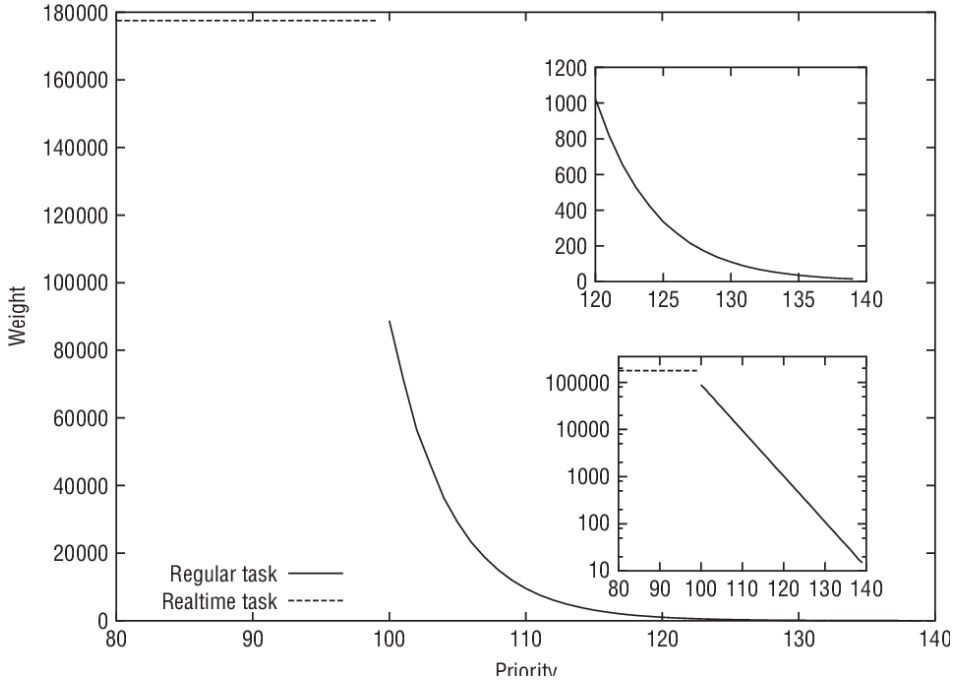


Figure 2.2: Priority to weight conversion [20]

Priorities aside, one of the main differences between tasks is the scheduling class to which they belong. The Linux scheduler is modular and nowadays capable of managing up to five different types of tasks in a preemptive approach, being these described next in Table 2.1.

Table 2.1: Scheduling classes of the kernel

Class	Description	Policy	File at kernel/sched
Stop	Special kernel thread	-	stop.c
Deadline	Hard R-T	SCHED_DEADLINE	deadline.c
Real Time	Soft R-T	SCHED_{RR,FIFO}	rt.c
Fair	Normal processes	SCHED_{NORMAL,IDLE,BATCH}	fair.c
Idle	Idle task	-	idle_task.c

Hard and Soft Real Time policies have a deadline. However, Hard policies are particularly restricted to finishing on time, while Soft RT have a more indicative budget. It can be observed that each of this families of tasks (designated "scheduling classes") is differently administered by the scheduler. This is, instead of managing every incoming task equally, using for example a FIFO algorithm, the main scheduler relies on specific "scheduling algorithms" for each family of tasks. A good exemplification of this modularity would be normal user processes. Those are managed by means of the well-known CFS 0(1) Ingo Molar scheduler [21] (p.15), that handles a balanced red-black tree data structure of task_struct* pointers.

As shown in the Figure 2.3, the kernel scheduler is divided into a main scheduler (`kernel/sched/core.c`) and a periodic one. The latter is called by the timer (timer.c) via interrupts with a frequency of certain HZs. This way, the periodic scheduler allows preemption and

other cyclic tasks, like the activation Load Balancer, that takes on an important role during those intermittent interrupts.

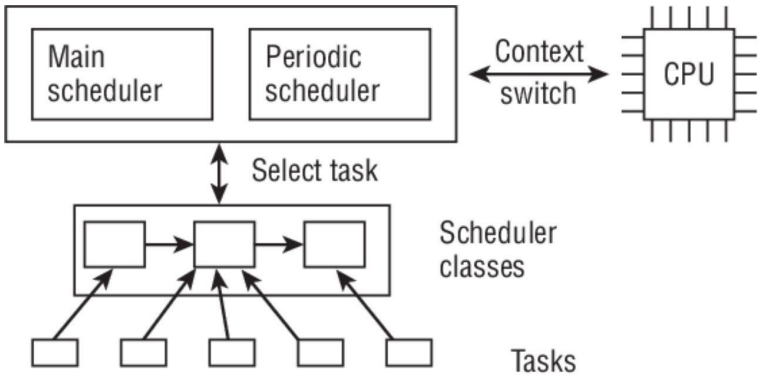


Figure 2.3: Overview of the main scheduler components [20] (p.87)

Note that there is a hierarchy in the order the classes are placed. Hitherto, such hierarchy was set up at compile time and therefore there was no mechanism to add a new scheduler class at runtime; this is something the project aimed to improve. There are other aspects related to the scheduler, such as group scheduling or scheduler entities [20], but those are not relevant to the subject at hand and therefore are not going to be covered.

However, we cannot allow ourselves to be complacent. Even being such a complex and efficient system, there is still **big room for improvement**. For instance, as this project tried to remedy, why can't developers create and plug-in their own scheduling classes according to their specific necessities? Perhaps -and this is just a hypothetical example- large-scale Cloud cluster systems that simultaneously train a specific machine learning estimator for weeks, such as a neural network [22], could have the scheduler optimized for that specific task, saving resources and time.

2.2 Relevant code snippets

Granted, a scheme as the one depicted above requires many macros, along with structure definitions: for the scheduler classes, tasks, special run-queues with specific spin-locks, and so forth. The definition of the struct *task_struct* alone, necessary for the manipulation of tasks, occupies more than six hundred lines of code in the kernel as of version 5.7.2.

Table 2.2 shows the most important files related to this topic, with the path relative to the kernel's main folder.

Table 2.2: Important kernel files for scheduling

File	Purpose
`include/linux/sched.h`	task_struct definition and scheduler API
`kernel/sched/sched.h`	run-queues and scheduling class definition
`include/uapi/linux/sched.h`	Scheduling policies macros, such as SCHED_NORMAL

File	Purpose
`kernel/sched/core.c`	scheduler and Load Balancer implementations

For all of this, and the constant mix of assembly and C language for specific low-level optimizations, reading the kernel code can sometimes be quite strenuous. In addition, there is no C standard library at this point. Understanding kernel code requires tons of practice so that you can abstract from the low-level details, that you already understood, and focus on the main idea behind the implementation. We believed it would be interesting to show **the entire process** from where the kernel calls the function `schedule()`, to the moment the scheduling class picks the next task, understanding that way both coding tools and workflow.

In this example the first step is written in assembly. At a recent Linux kernel version (5.1), in `arch/nios2/kernel/entry.S` we can find the following assembly label, called at the end of an interrupt or exception. The directory `arch/nios2` contains the required assembly code to port Linux to the Nios II 32-bit embedded-processor architecture, used in FPGAs.

```
Luser_return:
291     GET_THREAD_INFO r11
292     LDW     r10, TI_FLAGS(r11)
293     ANDI32  r11, r10, _TIF_WORK_MASK
294     BEQ     r11, r0, restore_all
295     BTBZ    r1,  r10, TIF_NEED_RESCHED, Lsignal_return
(...)
298     call    schedule
299     BR ret_from_exception
```

In line 291, a pointer to the thread information is stored in register r11, from which it will then extract the flags into r10. Then, an AND bitwise operation is performed to check whether we need to re-schedule (line 293) or not in which case it will restore (line 294). If necessary, the function schedule is called.

The function schedule is just a wrapper, that mainly disables and enables preemption. In between, it calls the main scheduler function that we will explain later.

```
asmlinkage __visible void __sched schedule(void){
    (...)
    preempt_disable();
    __schedule(false);
    sched_preempt_enable_no_resched();
    (...)
}
EXPORT_SYMBOL(schedule);
```

The asmlinkage modifier is used to make it possible to call the function from assembly files. The *EXPORT_SYMBOL* macro makes the function globally visible for built-in kernel code, and it will be available in the symbols table `/proc/kallsysms` with the *T* label, for it is

located in the code -text- section. In order to use it in a kernel module, it should also be visible in `/lib/modules/$(uname-r)/build/Module.symvers`, which for a module implies `make (...) modules_install`. Anyway, you can always use the function address, thanks to having a monolithic kernel where all its parts share an address space.

The function `___schedule()` is only called when there is explicit blocking (due to a mutex, a semaphore, a waitqueue…) or the TIF_NEED_RESCHED flag is set, when checked on interrupts and user-space return paths, like in the example shown below. Be aware that the following code is not the complete function, since we have deleted the parts we do not consider of relevance for this example and also added all the comments to make it apprehensible.

```
static void __sched notrace __schedule(bool preempt){
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;
```

Variables *prev* and *next* are respectively the previous and following task running on the calling CPU. The run-queue flags used to manipulate the lock are *rf*, the pointer to the processor's run-queue is `*rq` and variable *cpu* is the identification number of the processor.

```
    cpu  = smp_processor_id();
    rq   = cpu_rq(cpu);
    prev = rq->curr;
    local_irq_disable();
    rq_lock(rq, &rf);
```

The number of the current CPU is acquired, as well as the processor's run-queue. Then, the current task running is retrieved, interrupts are disabled and the run-queue spin-lock is acquired.

```
    next = pick_next_task(rq, prev, &rf); /* Pick next task using a class    */
    clear_tsk_need_resched(prev);         /* Clear previous task flags       */
    clear_preempt_need_resched();         /* Clear flags                     */
    if (likely(prev != next)) {           /* If different tasks              */
        rq->curr = next;
        /* Also unlocks the rq: */
        rq = context_switch(rq, prev, next, &rf); /* Performs context switch */
    }
    (...)
    balance_callback(rq);
}
```

`likely()` and `unlikely()` are compiler optimizations to inform that most likely the jump will -or will not- be taken after the condition evaluation, and perhaps exploiting Instruction Level Parallelism. As we can observe, the previous function finally calls `pick_next_task()`, passing the processor's run-queue and the previously executed task (as well as some flags we are not interested in).

This is one of the most important functions for the scheduler yet easy to understand. Here it is shown as it can be found in a recent kernel version (5.2).

```
/*  Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf){
    const struct sched_class *class;
    struct task_struct *p;
    if (likely((prev->sched_class == &idle_sched_class
        || prev->sched_class == &fair_sched_class) &&
        rq->nr_running == rq->cfs.h_nr_running)) {
```

The above condition will be true if the previous thread running was idle or fair and all the threads in the core's run-queue are property of the class Fair. This is the most common situation in non-critical systems.

```
        p = fair_sched_class.pick_next_task(rq, prev, rf);
        if (unlikely(p == RETRY_TASK))
            goto again;
        if (unlikely(!p))
            p = idle_sched_class.pick_next_task(rq, prev, rf);
        return p;
    }
```

This code is an optimization, for if all the tasks are from the fair class there is no need to keep looking. If that fails, it will just iterate over the entire list of scheduling classes until success. This is guaranteed to happen at some point, since the Idle class never rejects an incoming task and it always have an idle process waiting to run.

```
again: /*
for_each_class(class)= for(class=sched_class_highest;class;class=class->next)*/
    for_each_class(class) {
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
    BUG();
}
```

In the code shown above, the scheduler needs to pick the task that will run next in a certain CPU (the one that called this function). If all the tasks in that core are from the Fair class then it directly calls the `pick_next_task()` from the Fair scheduling class. The classes return NULL if there is no task of their property in the invoking core's run-queue. It iterates over all the possible classes, stored in a linked list, and try one by one to receive a valid next task. If after picking the next task, a new process with higher priority appears, the previous one

will be preempted as soon as possible. If no class success, the function calls BUG(), which is defined as an invalid assembly instruction, which means the CPU will throw an invalid opcode exception. The OS will undoubtedly kill the thread.

The complexity of the functions for picking the next task of each scheduling class will depend on the implemented algorithm, and most likely it will not have much in common with the other scheduling classes´ way of dealing with it. There are, nonetheless, two things we could state all the scheduling classes share and will have to do:

1. They will need to link their implementation to the scheduler class generic functions' pointers.

```
const struct sched_class example_sched_class = {
   .next               = &next_sched_class,
   .enqueue_task       = enqueue_task_example,
   .dequeue_task       = dequeue_task_example,
   .pick_next_task     = pick_next_task_example,
   (...)
};
```

2. They will create their own featured run-queue and store it on the processor's run-queue. The only exceptions to this rule are Idle and Stop classes. For instance, the red-black tree run-queue for the CFS scheduler defined at line 488 of `kernel/sched/sched.h`, that we can see retrieved in the following line of code.

```
struct cfs_rq *cfs_rq = &rq->cfs; /* Obtaining the CFS tree */
```

After the reading of this chapter, we expect the reader to be familiar with the most important tools employed in the Linux scheduler and its general workflow. Similarly, he/she should be aware of the motivation of this project.

Much more information about the Linux kernel can be found online. A good suggestion is the book [20], which was of great help, and is probably one of the standard manuals when it comes to the Linux kernel quirks. Nevertheless, be aware that it is a little bit outdated. For instance, on page 68 you will read *"…Linux does not support hard real-time processing, at least not in the vanilla kernel…"*, and that is not true from version 3.14.1, as we have learned in this chapter. At the end of the day, being outdated is the norm when it comes to the previous Linux kernel documentation, as it is such an evolving subject.

Chapter 3

Prior work and LITMUS-RT patches

In this chapter we cover prior work conducted in the Systems area related to the development of scheduling algorithms and the contributions made to related open-source initiatives during this project.

3.1 The LITMUS-RT project

A significant amount of work has been priorly conducted in the Systems area related to scheduling algorithms prototyping. Some researchers have attempted to force thread migrations and task dispatching by means of system calls. However, this approach might introduce too much overhead [6]. Moreover, there are many kernel related aspects and hardware events that *user-space* designs cannot be aware of.

Researchers at the University of North Carolina [5] developed their own version of the Linux kernel called LITMUS-RT, (*Linux Testbed for Multiprocessor Scheduling In Real-Time System*), which is currently maintained by members of the **Max Planck Institute for Software Systems** (MPI-SWS). The main goal of the project was to enable the implementation of several real-time algorithms on Linux, as well as collecting their associated performance statistics afterwards.

For that purpose, they included real-time (RT) algorithms, referred to as "plugins", into the Linux kernel. These algorithms were variants of EDF and P-Fair. They also developed a *user-space* library named Liblitmus, which provided several wrapping tools for managing the aforementioned time-constrained threads. With that library, developers are allowed to visualize the implemented scheduling plugins, and likewise switch between the currently available ones. More importantly, with Liblitmus developers are capable of testing the algorithms, by creating specific tasks of the currently activated scheduling plugin by means of kernel hooks and callbacks.

In their website [23], LITMUS-RT is defined as follows:

"*LITMUSRT is a real-time extension of the Linux kernel with a focus on multiprocessor real-time scheduling and synchronization. The Linux kernel is modified to support the sporadic task*

model, modular scheduler plugins, and reservation-based scheduling. Clustered, partitioned,and global schedulers are included, and semi-partitioned scheduling is supported as well."

The most important aspect of all the kernel changes introduced by the LITMUS-RT project was related to the simultaneous administration of several new plugins. To achieve this, they prepared a generic scheduling class, *litmus_sched_plugin*. The new class performed only tracing and trivial real-time tasks, things that any RT algorithm should, while delegating the specific details of implementation to each of the included scheduling plugins. It was, in that sense, similar to the file *core.c* explained in the previous chapter, as it performed the most generic operations to later devolve to specific scheduling classes. Hence, when the Litmus class is called it performed all the generic real-time preliminaries.

The ingenious techniques and modularity that the LITMUS-RT project demonstrated were important to us, as they helped to better learn and carry out this project. However, it is important to point out **two important limitations** that the project still had, and that differentiated it from the FSAC kernel or PMCSched.

1. The litmus scheduling class supports real-time generic tasks. Hence, the plugins included in the framework cannot be of any other nature, thus significantly reducing the number of possibilities to test. On the other hand, we intended FSAC and PMCSched to be frameworks for all types of scheduling algorithms.

2. Due to the way LITMUS-RT is implemented, it is still necessary to re-compile the kernel any time a new plugin is created, as they did not support runtime scheduling modules insertion. Consequently, **LITMUS-RT was also not dynamic**. On the contrary, it was our hope to make FSAC that way. As already explained in the introductory chapter, we wanted to achieve that dynamism in order to save hours of kernel recompilation and avoid having to reboot the system afterwards.

We considered that making LITMUS-RT a generic framework for scheduling algorithms (this is, turning LITMUS-RT into our idea of FSAC) was a particularly challenging goal due to the first limitation. After all, LITMUS-RT' own nature was focused on real-time scenarios. On the contrary, the second limitation could be fixed and patched by us, while gaining in return some experience for our own software. Thus, we submitted a patch. But before discussing that, it is important to understand how does LITMUS-RT work.

We will now ellaborate on code-related aspects of the LITMUS-RT project. Several things done in the LITMUS-RT project are worth mentioning, as they constitute interesting hints for this project's goal. In fact, some of their ideas were "recycled" during the development of FSAC.

As explained in the previous chapter, the Linux scheduler manages a linked list of scheduling classes. Granted, they had to add their generic scheduling class to a list of classes, as can be seen below. They added their scheduling class into the first position, which was probably not the least intrusive approach, as other threads with higher priority are relegated.

```
1311 extern const struct sched_class litmus_sched_class;
1312 extern const struct sched_class stop_sched_class;
1313 extern const struct sched_class dl_sched_class;
```

```
1314 extern const struct sched_class rt_sched_class;
1315 extern const struct sched_class fair_sched_class;
1316 extern const struct sched_class idle_sched_class;
```

In addition, they had to find a way to allow the developer to interact with the kernel-level scheduler. As it is usual when dealing with the kernel, they opted for the `/proc` file-system, adding several files in it. The most important of them all was `/proc/litmus/active_plugin`, which is a fairly self-explanatory name.

The user-space library Liblitmus can be found at GitHub, and needs to be compiled against the LITMUS kernel source code. Hence, if someone is willing to test this kernel, we suggest not to remove the kernel files after installation, and to move the executables to `/usr/local/bin`. In order to make things simpler to the user, they included the `setscheduler` command line tool, which wraps the interaction with this special file. The user can check which scheduling plugins are available by reading from the `/proc/litmus/plugins/loaded` file, and by then performing the scheduling class change. This is the line of the bash script where the change is requested, after checking that the desired scheduling plugin exists:

```
ADIR=/proc/litmus/active_plugin
PDIR=/proc/litmus/plugins/loaded
CHOICE=$1
(...)
echo "$CHOICE" > $ADIR 2> /dev/null || \
    echo "Error: Cannot write to $ADIR. Do you have root privileges?"
```

If the user or the bash script attempts to echo in the `active_plugin` file with a scheduling plugin as argument, the write system call will be executed and handled, with the function `litmus_active_proc_write()`. This way, the kernel function can safely read the user message stored at the array pointed by `buffer*` and act accordingly. If the plugin name is found in the linked list, it can be switched calling `switch_sched_plugin()` that then writes in the dmesg log via `printk()`.

As a matter of fact, the function `switch_sched_plugin()` does not do much but calling the function `stop_cpus()`, which is a little more tricky as it receives a callback to function `do_plugin_switch()` apart from the plugin and the affinity mask.

```
 err = stop_cpus(cpu_online_mask, do_plugin_switch, plugin);
```

Function `stop_cpus()` is just a wrapper. The function `___stop_cpus()` could also be very simple: Kill all the tasks running on each processor and then switch to that specific real-time plugin. Nevertheless, that approach -apart from being quite aggressive- would not suit them, as the tasks the scheduler will be dealing with are real time. Therefore, it was essential that the processors were well synchronized. For that reason, each and every processor will be stopped and finally the global variable litmus changed. This variable (struct of function pointers) is then used to call the active plugin from the litmus scheduling class. Kernel preemption needs to be disabled during the entire queuing process, just in case this code itself gets preempted by a stopper kthread, which might wait for other stoppers, resulting everything in a fatal deadlock. After disabling preemptions for every available CPU, the

tasks will be stopped and the new thread queued. On each targeted core, the function will be executed with the highest priority, preempting any task of the CPU and "monopolizing" it.

Again, LITMUS exhibits a quite aggressive approach. This is probably one of the reasons the LITMUS-RT's developers decided to include the litmus scheduling class in the first position of the linked list, as the user will have the certainty that, after executing the command, the cores will be working on their algorithms right away. This facilitates the analysis of statistical data the OS presents the user, discussed later. The callback function passed as the next parameter is what all the cores execute, and it is very simple yet important (`do\_plugin\_switch()`). It changes the pointer *litmus*, which indicates which plugin is the loaded one, finishing the process. Once the plugin scheduler is updated, in order to test the algorithm new tasks of the litmus scheduling class need to be created. For that purpose, Liblitmus includes the `rt_launch` bash command.

Thanks to the study of their source code (published at GitHub) and the documentation available on their website, we could spot some interesting ideas and realize under what means they handled certain situations. A very common practice in Linux kernel development is using the `/proc` file-system to expose information about the processes, so we were not surprised when saw that was precisely what they did. Nonetheless, they also provided within their library with some user-mode tools such as scripts that wrapped the `/proc` interaction, which we acknowledged useful.

The LITMUS-RT project also includes some **user-space command line tools** to perform statistical analysis at runtime of the performance of their real-time scheduling algorithms that will be explained now. We got familiar with their usage and implementation, as such features could be of great help for developers in frameworks for runtime plugins as the ones we wanted to create. Feather-Trace tools, part of the LITMUS-RT project, are also useful to record how long a scheduling decision (SCHED_START, SCHED_END,…) or when a context switch takes place as well as statistical measures from generated CSV-files. Those are usually measured in cycles, where for instance a 1.70GHz CPU will do 1700 cycles per microsecond. In particular, we found *rtspin* to be of great interest. It creates loop tasks used to test CPU workload. These tools provided valuable insights.

For example, the following command executes the task for 5 seconds on CPU 0, with a WCET (Worst-Case Execution Time) of 1.5 ms:

```
$ rtspin -p 0 1.5 10 5
```

On the other hand, Sched-Trace tools are used to record which tasks are scheduled at what point with `st-trace-schedule` after using `rtspin`. Events captured include task dispatching (switched to), preemption (switched away), suspension (blocked), resume (wake-up). Additional time-related information is provided, extracted from the PCB task_struct. Other developed tools are `st_dump` for a human-readable version of the .bin, `st-draw` for the trace in .pdf format and `st_job_stats` for generation of a CSV reports.

3.2 The main LITMUS-RT patch

As stated earlier, LITMUS-RT did not allow us to load new scheduling algorithms at runtime; LITMUS-RT did not handle the insertion of those plugins as modules. This was one of the biggest limitations identified before in section 4.1. In order to understand how to address that, we decided to start by writing our own version of a scheduling plugin according to the LITMUS-RT API.

When finished coding, we still encountered the same old issue of having to recompile the kernel with the new plugin every time an error was detected, as expected. It took almost half an hour -in a system with good specifications, an octa-core Intel- to conclude each compilation. In a regular laptop that same task could have easily lasted several hours.

This is why we decided it would be a good idea **publish a patch for the LITMUS-RT public source code**, so that we could load our modules at runtime on their kernel. That was similar to what we wanted to do for this project on the first place (FSAC), but without having to concern about a lot of related aspects just yet and for a kernel solely focused on real-time scheduling.

Their repository was forked and we started with the development. It was not as trivial as we first expected. We ended up having to modify several source files and submitting dozens of commits, as shown in Figure 3.1. Besides adding support for plugins unregistering, we had to consider the potential errors due to plugin-switching while the CPU is busy performing some previous real-time tasks, and for that we had to employ a reference counter, a common practice in drivers development.

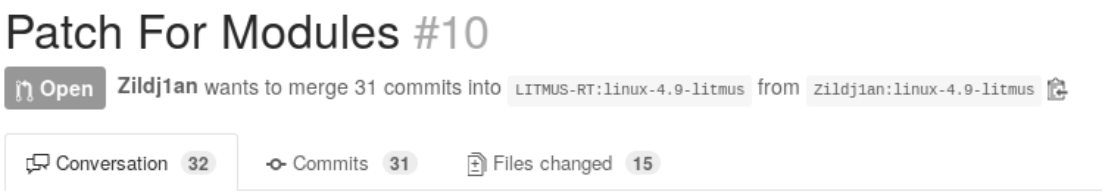


Figure 3.1: The main patch in numbers

Coding for the LITMUS-RT kernel implies much more than originally expected. For example, it was mandatory to follow a strict coding style and specific software design patterns broadly used on the system, as the maintainer explained as: "*Thanks a lot for the revision. Before merging this, could you please make a pass through the diff to make the new code follow Linux coding style in all places? Thanks.*". In the kernel development, not anything that works, works.

We will now briefly discuss the implementation details of this patch. The first function added right away was the one that allowed the unregistering of the plugins after searching in the plugin linked list. It was important to use the EXPORT_SYMBOL macro so that it could be called by modules at runtime. We also had to add that macro to all the other LITMUS-RT functions called in the rest of the plugins, as we planed to call them at runtime on ours.

```
int unregister_sched_plugin(struct sched_plugin* plugin) {
    int unregister = 0;
```

We needed to make sure that the plugin the developer is trying to unregister is not the active one, which would otherwise have devastating consequences.

```
    if (strcmp(litmus->plugin_name, plugin->plugin_name) != 0) {
        unregister = 1;
        raw_spin_lock(&sched_plugins_lock);
        list_del(&plugin->list);
        raw_spin_unlock(&sched_plugins_lock);
    }
    else
        printk(KERN_ALERT "The currently active plugin cannot be removed\n");
    return unregister;
}
EXPORT_SYMBOL(unregister_sched_plugin);
```

We also had to add exit functions for all of the scheduling algorithms previously implemented by them, so they could be removed as normal modules. Reached that point, the biggest problem of the patch we created was that, without a careful revision of the number of references, situations like the one depicted below by the LITMUS-RT creator and main maintainers (Figure 3.2) could lead to kernel panic scenarios.

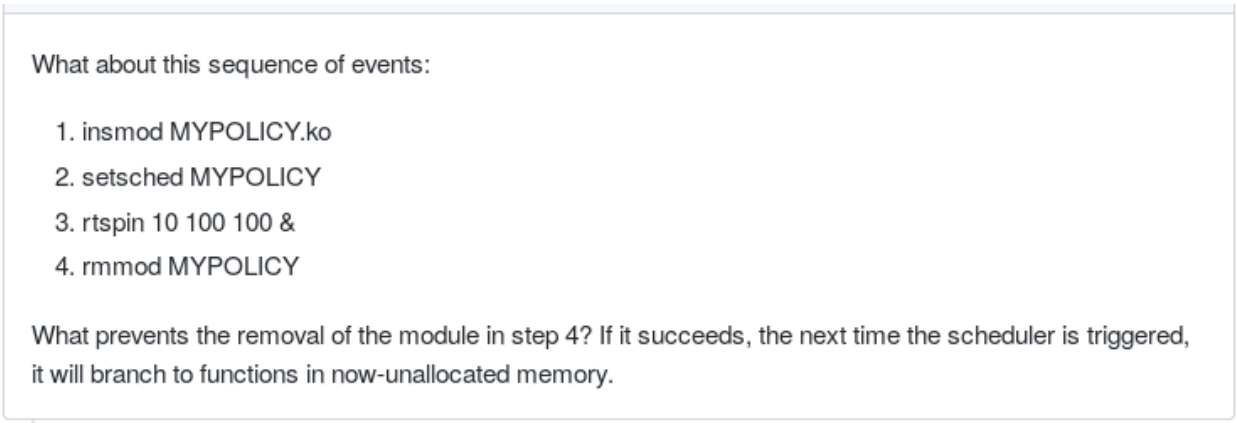


Figure 3.2: Potential error scenario of plugin switch

This is why we had to take care of the aforementioned reference counter in the functions related to the activation and deactivation of the plugins. It was a **similar case to the one faced when writing drivers**, as they cannot be unloaded until every agent is done using them. The solution in both cases is to locate the reference counter on the functions in charge of activating and de-activating, the plugin, in this case.

Shown below is an example of the most simple activation and deactivation function written in the patch for any of the scheduling plugins, using them to avoid the plugin to be removed from the special linked list at the module exit function. The function *nuevo_deactivate_plugin()* was then in charge of decrementing the reference counter as shown below.

```
static long nuevo_activate_plugin(void) {
    /* Maybe some extra work in the future here */
    try_module_get(THIS_MODULE); /* Increase counter */
    return 0;
}
static long nuevo_deactivate_plugin(void) {
    module_put(THIS_MODULE); /* Decrease counter */
    return 0;
}
```

We finally managed to complete the patch and successfully loaded our plugin into the new kernel in a safe manner. Again, it was only for real-time algorithms, but was a good step towards the direction of this project, and a nice contribution to an interesting open-source project.

3.3 Other maintenance patches

Apart from fixing some errors in the project that were committed and accepted by the project maintainers, there was an important load of maintenance patching. This resulted from problems we faced when compiling the LITMUS-RT kernel, due to the lack of code maintenance. For instance, when compiling with a new version of the GCC compiler (v.81) new warnings arised, which prevented us to complete the compilation of the Linux bzImage (big zImage).

That bug was reported in 2017 by Arnd Bergmann in the Linux kernel mailing list (LKML). All the community kernel patches that we discuss can be found at [24]. Although we were lucky in finding a solution to this issue, we had to dig much more in the mailing lists and forums in search for suitable solutions. Other problems we had to solve that are worth mentioning are as follows:

1. `R_X86_64_PLT32` not handled error when making the bzImage, patched in 2018 by Linus Torvalds and H.J. Lu at patch with Subject "x86: Treat `R_X86_64_PLT32` as `R_X86_64_PC32`".

2. Similarly, we had to take care of the warning *restrict* that prevents kernel compilation, patched in 2018 by Sergey Senozhatsky at patch with Subject "*[PATCH] tools lib subcmd: do not alias select() params*".

   ```
   From: Sergey Senozhatsky <sergey.senozhatsky@gmail.com>
   Cc: Matthias Kaehlcke <mka@chromium.org>
   Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>
   Subject: [PATCH] tools lib subcmd: do not alias select() params
   (...)
   ```

Unfortunately, the main LITMUS-RT maintainer (Prof. Björn Brandenburg) explained -as shown in the following screenshot 3.3- that trying to keep the old LITMUS-RT version up-do-date with the most recent libraries was a futile work. He asserted that the only way

that LITMUS-RT could continue being suitable for the newest upgrades was by porting the kernel.

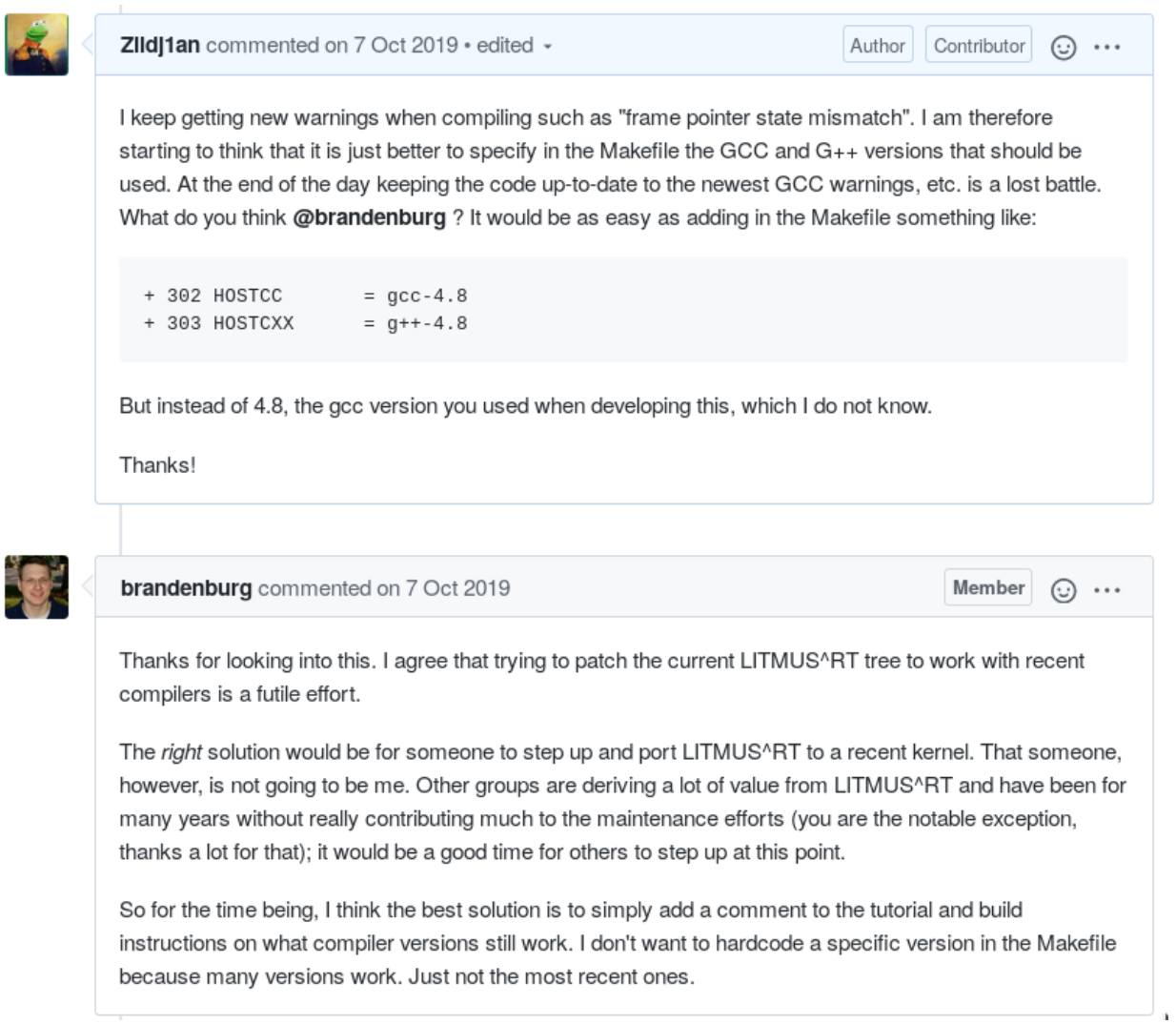


Figure 3.3: A new version of LITMUS-RT needed

We did not stop contributing to the LITMUS-RT project when done studying it. For instance, August 15 Prof. Brandenburg merged our commit "*Add missing functions for the dummy litmus plugin*" (patch *Zildj1an:patch-9*, pull request *#18*).

Chapter 4

Development of the FSAC kernel

In this chapter we will provide a high-level description of the implementation and main concepts behind the FSAC kernel. In the first subsection we will discuss what features should be included, by outlining a hypothetical scheduling algorithm. After that, we will describe the API developed for FSAC. Finally, the internal modifications of the Linux kernel will be covered. Among these changes, we included support for statistical analysis of the plugins and support for real-time designs.

The FSAC (*Framework for Scheduling Algorithms Creation*) kernel contributes to the academic research on the Systems area and the Linux community. It is a variant of the Linux kernel in development stage. Inside the Linux kernel 4.9.30 stable version [25], we have included a complete framework for scheduling algorithms that allows a rapid prototyping. Until now, it was a necessity to recompile the kernel, a tedious task that consumes many resources and time [20]. It is important to emphasize that we compile and reboot the system without having the certainty that we will not encounter errors during or after initialization. Moreover, as a scheduling framework, FSAC performs all the repetitive tasks that every algorithm has to, no matter their nature. In essence, this ease things to OS developers and can surely save a few headaches thanks to an intuitive API we describe in this chapter.

In addition, FSAC includes **support for real-time algorithms** by means of some extra modifications inside the kernel that we will be cover in Section 4.4. By any means this implies algorithms have to necessarily be real-time. With this framework, researchers and developers could easily implement scheduling algorithms within a loadable kernel module without concerning about several low-level OS issues, and then actually run their algorithms while testing their performance.

Studying LITMUS-RT [5] was a great way to extract ideas and produce some new code -in the form of patches- that could in return be added to this final project and help designing the FSAC kernel. As a matter of fact, some ideas that were discussed in Chapter 3 are closely related or served as inspiration to several abstractions that we will present in this chapter.

4.1 Outlining an algorithm with FSAC

A good first step in order to write a prototyping framework for scheduling algorithms is to write a draft of a hypothetical algorithm, assuming that we have all the necessary support from the kernel to plug it. We did that so we did not have to worry about neither low-level related issues nor doing an algorithm belonging to a specific class that could not be inserted at runtime. This is what used to happen with the LITMUS-RT kernel (view Chapter 3), where the scheduler was limited to real-time algorithms priorly pre-linked into the list the kernel manages. We will provide an outline of the drafting of this hypothetical algorithm in this section.

In order to add our hypothetical algorithm at runtime into the Linux kernel, we will use Linux loadable kernel modules. Modules are special code blocks whose object files can be loaded at runtime, in order to help developers deal with some of the disadvantages of a monolithic kernel, like in the case of the SMP Linux kernel. By writing a hypothetical scheduling algorithm with a kernel module, we will be able to discover what would be necessary in order to make the framework work. This is, we will be able to discern **what features will the developer demand**.

The first code needs to contain the minimal necessary tools to make the module function, which includes an initialization function where the plugin could potentially be included at runtime into a list of available algorithms. To link all the functions from this plugin to the generic scheduler we will also need a structure of pointers to functions as callback mechanism. Finally, we would need to include two basic functions that should be present in any scheduling algorithm, one to schedule new processes and other to admit task into its run-queue. Similarly, we would need to link the recently developed functions to the structure's hooks that the scheduling plugins should have.

As the algorithm is supposed to be plugged and unloaded at runtime, we also need an exit function just in case the plugin is removed. Furthermore, we need to be careful about an user accidentally removing a module that is being used to perform some scheduling decision in a core, as this could be a complete disaster for the system. This is similar to the scenarios a driver usually faces. Hence, to take care about this situation we need to add two more functions and include them into the list of registered functions; An activation function, that will increase a module counter that prevents the removal when the plugin is chosen as the current scheduling plugin, and a deactivation function, that will decrease the aforementioned counter.

Every processor will need to store the information required to perform the scheduling decisions and the mechanisms for the plugin to be retrieved must be designed. In this way, when the scheduling function is executed, PMCSched can be up-to-date with the core situation. This involves -at least- a pointer to the more recently scheduled task, the number of the core where the decisions are taken place and a custom queue with the runnable tasks. We will also need to create a function to free all that memory at the module removal, whose code is not relevant now but can be consulted at the source code of this project. Every processor in the system should also include its identification number, among the other variables previously mentioned:

```
struct cpu_state { /* CPU state */
        int cpu; struct task_struct* scheduled;
        spinlock_t queue_lock;
        sized_list_t fcfs_queue; /* FCFS-queue  */
        struct list_head link_queue;
};
```

We also need to add a per-processor allocation macro, DEFINE_PER_CPU, to statically allocate this state required for the algorithm. Every processor should also initialize its list.

```
static DEFINE_PER_CPU(struct cpu_state, cpu_state);
```

We cannot initialize the linked list at the module insertion, as each processor will have its own queue. Therefore, we need to initialize the queue at the activation function, were we can also give each core a number for the state. We will also define the macro `cpu_state_for()` to extract the CPU's structure state depending on the id, and `local_cpu_state()` for retrieving the state of the CPU running the code at that moment.

```
#define cpu_state_for(cpu_id)   (&per_cpu(cpu_state, cpu_id))
#define local_cpu_state()       (this_cpu_ptr(&cpu_state))

static long nuevo_activate_plugin(void) {
    int cpu; unsigned long flags;
    struct cpu_state *state;
```

In the activation function we iterate over every available core, initializing its variables.

```
    for_each_online_cpu(cpu){
            printk(KERN_INFO "Initializing CPU %d...\n",cpu);
            state = cpu_state_for(cpu);
            state->cpu = cpu;
            state->scheduled = null;
```

In addition, when we initialize the run-queue we should be careful with deadlocks, and thus we acquire the lock and disable IRQs.

```
            spin_lock_irqsave(&state->queue_lock, flags);
            init_sized_list(&state->fcfs_queue,offsetof(
                struct fcfs_queue_node,links));
            spin_unlock_irqrestore(&state->queue_lock, flags);
    }
    try_module_get(THIS_MODULE); /* Increase counter */
    return 0;
}
```

Reached this point, the algorithm is prepared to consider too **changes in the state of a task**, as there as many scenarios not considered yet. For instance, in reality tasks usually go to the I/O waiting state. In addition, schedulers need to handle new tasks and the resuming ones after the self-suspension. All of this cannot be avoided even for the most basic version

of the algorithm. Some function has to be created in order to handle the exit of a task. Both of them have to be then included into the `sched_plugin` function pointers' association struct. Other function for resuming tasks (changing back to TASK_RUNNING) should also be considered. For now, functions related to preemption will not be covered.

For testing purposes, it is also interesting to add some **interaction possibilities** in FSAC with the `/proc` virtual filesystem, so that the developer can easily test the state of the cores at each moment of the execution. For that purpose, we will declare a `proc_dir_entry` that will be initialized at insertion and erased at removal from the virtual file system proc. Also, a read system-call must be handled by the module so that the developer can do `$cat /proc/my_algorithm` and check the information. That system-call handler will be linked to a struct of pointers to functions as callback mechanism, just like with the algorithm development API.

```
static struct proc_dir_entry *my_proc_entry; /* My /proc file entry */
```

The basic operations that the module can handle will only include reading from it to retrieve the core's relevant information:

```
const struct file_operations fops = {.read = nuevo_read,};
```

The framework, along with providing support for this whole hypothetical algorithm, will need to modify the kernel scheduler itself and add some tools to easily execute and study tasks for the new plugin family. But before that, it is already possible to spot what is required in order for a basic plugin to be created:

1. A **structure with functions that every plugin algorithm should implement**, that will be linked to the new ones made by the developer. We saw such abstraction being used at the begging of this outlining.
2. A **list of scheduling plugins** where the new plugin can be dynamically added at insertion and removed when demanded. Before being added, some security checks have to be made. For example, it would be useful if the framework checks whether or not has the developer implemented all the required functions, so no null pointers are invoked at runtime.
3. Some wrappers in order to make the development easier could also be added. For instance, for managing the `/proc` filesystem easily, printing debugging messages or managing tasks' queues and their respective spin-locks. In addition, a function for retrieving the CPU number for a given task would be interesting.

4.2 Developing the FSAC API

The previous section 4.1 involved developing a hypothetical scheduling algorithm using a Linux loadable kernel module. The main goal was to highlight the tools and features a developer implementing a scheduling plugin would require. In order to do so, and for this first and most basic version of the framework, only the most basic topics regarding scheduling where covered, and no special consideration was given to complex preemptive situations,

possible tools for real-time algorithms, and so forth. Perhaps, all those could be added in some future version(s). Nevertheless, in order to make that possible we should try to the keep the creation of the API as modular as possible.

The first thing we noticed in the previous section that the developer would need is a **set of pre-defined functions** in a structure *sched_plugin*, related to the process of tasks' scheduling, whose pointers could be linked to the developer's module functions, as hooks. That could also be useful for security-related aspects (i.e. the framework can check that the module contains all the minimal required functions), as well as making possible the runtime insertion, as the value of those function addresses can be dynamically updated in order to point to a newly inserted module. A header file including all the generic scheduling plugin can be found inside `/include/fsac_plugin.h`. This file should be the header of any plugin developed with FSAC.

```
#ifndef _LINUX_FSAC_PLUGIN_H_
#define _LINUX_FSAC_PLUGIN_H_

typedef long (*activate_plugin_t) (void);
(...)
typedef long (*admit_task_t)(struct task_struct* tsk);
```

Once all the function pointers are defined we can include them as fields of a structure, along with other values such as the plugin name and a variable to make the insertion or deletion into a linked list possible.

```
struct fsac_plugin {
        struct list_head  list;
        char *plugin_name;
        activate_plugin_t  activate_plugin;
        deactivate_plugin_t deactivate_plugin;
        schedule_t  schedule; admit_task_t admit_task;
        block_task_t block_task; task_new_t task_new;
        task_wake_up_t task_wake_up; task_exit_t task_exit;
        (...)
} __attribute__ ((__aligned__(SMP_CACHE_BYTES)));
```

We also need to add a pointer to the active scheduling plugin.

```
extern struct fsac_plugin *fsac;
```

Finally, we declare the dummy default plugin and a function that will be used to register new scheduling plugins dynamically.

```
extern struct fsac_plugin fsac_sched_plugin;
extern int register_sched_plugin(struct fsac_plugin* plugin);
(...)
#endif
```

This generic set of functions will be of help for any scheduling algorithms developer without

big prior experience. The pointer *fsac* points to the current scheduling plugin, the activated one. That, by default, will be the dummy plugin *fsac_sched_plugin*, that will be developed in the next section as now we are interested in explaining the framework from the developers perspective. In addition, some functions are provided in the FSAC source code, at `/include/fsac_list.h` to ease the management of the special kernel doubly-linked-lists.

Other problem that arises when dealing with a custom scheduling algorithm is the handling of the possible system-calls, required to perform a dynamic analysis of the performance of the plugin. At the end of the day, a new scheduling algorithm is of no use if its capacity cannot be tested under different workloads. Instead of making any developer deal with the `/proc` file-system by their own, like we had to do when outlying a hypothetical algorithm, the framework itself could add some support for the statistical analysis. We will cover this on the next subsection.

4.3 Support for statistical analysis on FSAC

After uploading the scheduling algorithm, the developer would most likely be interested in reviewing the status of the CPUs queues or any other information that he/she has stored in the per-cpu states and wants to keep track of. The `/proc` virtual file-system can be of help in these cases, as the developer can prepare with entries the handling of read or write system-calls. As adding a file for each uploaded plugin could be a little bit messy, we thought it would be interesting to add in FSAC a folder at `/proc` (see Figure 4.1), and inside the directory some extra files such as *loaded* that could retrieve the list of loaded plugins.

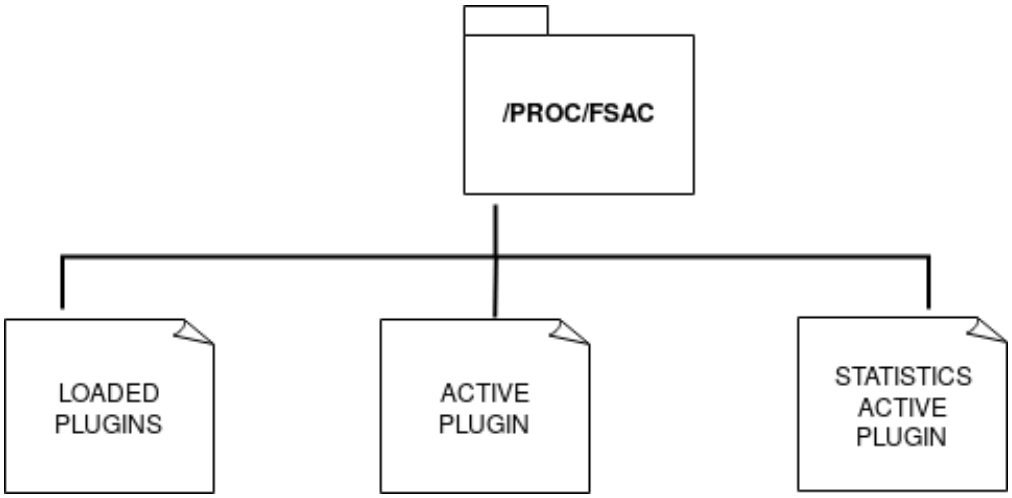


Figure 4.1: Interaction with FSAC via /proc

Following the same pattern, it would also be interesting to add a function *plugin_read* that every plugin module would have to implement in order to make the analysis possible. The framework could be in charge of managing all the low-level details of the virtual file-system so that the developer only needs to worry about filling the array of char that will be displayed. This function can be added into the structure *fsac_plugin* described in the previous section 4.1 of this chapter. Subsequently, we can include a function in the new source file `fsac_proc.c`,

where all the support for the FSAC Directory previously depicted will be included as well. Let us go through the more important pieces of this code for **dynamic statistical analysis support**.

1. At the beginning, we will just have a bunch of null pointers for proc entries that will be later initialized, as we can see in the code below. File `fsac_proc.c` purpose is to change the active FSAC plugin, as well as reviewing the list of registered plugins and the active one.

```
#include <fsac/fsac_proc.h>
static struct proc_dir_entry *fsac_dir = null, *loaded  = null,
*stats_active = null;
```

3. The active plugin can be switched simply by doing `$echo PLUGIN\_NAME > /proc/active`. The `proc_find_node()` function is included in `/include/fsac_proc.h` and it returns a pointer to the structure or the `list_item` depending on the first parameter.

```
static ssize_t active_write(struct file *filp, const char __user *buf,
                size_t len, loff_t *off) {
   if (found = proc_find_node(0,name,&proc_loaded_plugins)){
            if(err = switch_sched_plugin(found)) {
   (...)
}
```

4. When the developer wants to review statistics regarding the active plugin, he/she just needs to do `$cat /proc/stats_active`, and this function, as can be seen above, will invoke the per-module function of reading. In this way, the scheduling algorithms developer does not need to worry about things related with this process such as security issues or the correct handling of the system-call.

```
static ssize_t stats_read(struct file *filp,
               char __user *buf, size_t len, loff_t *off) {
    (...)
    read = fsac->plugin_read(kbuf);
    (...)
}
```

When outlying the hypothetical plugin at the beginning of this chapter, one of the things we included was a function to inform the system that a scheduling decision was just made. Adding a preemption state machine into FSAC could be of great help, particularly when it comes to handling SMP configurations and tasks migrations. With it, the FSAC scheduler could also check if a scheduling decision is still valid after performing a Context Switch. Let us enumerate the possible scheduling states we included in FSAC, each core containing one instance of the variable:

1. **TASK_SCHEDULED** ($1 \ll 0$): The currently scheduled task is the one that it should, and the processor does not plan to invoke *schedule()*.

2. **SHOULD_SCHEDULE** (1 « 1): A remote processor has determined that the processor should reschedule, but this has not been communicated yet (IPI pending)
3. **WILL_SCHEDULE** (1 « 2): The processor is currently executing `schedule()`, has selected a new task to schedule, but the Context Switch is still pending.
4. **TASK_PICKED** (1 « 3): The processor has not yet performed the Context Switch, but a remote processor has already determined that a higher-priority task should be the chosen one, after the task was picked.

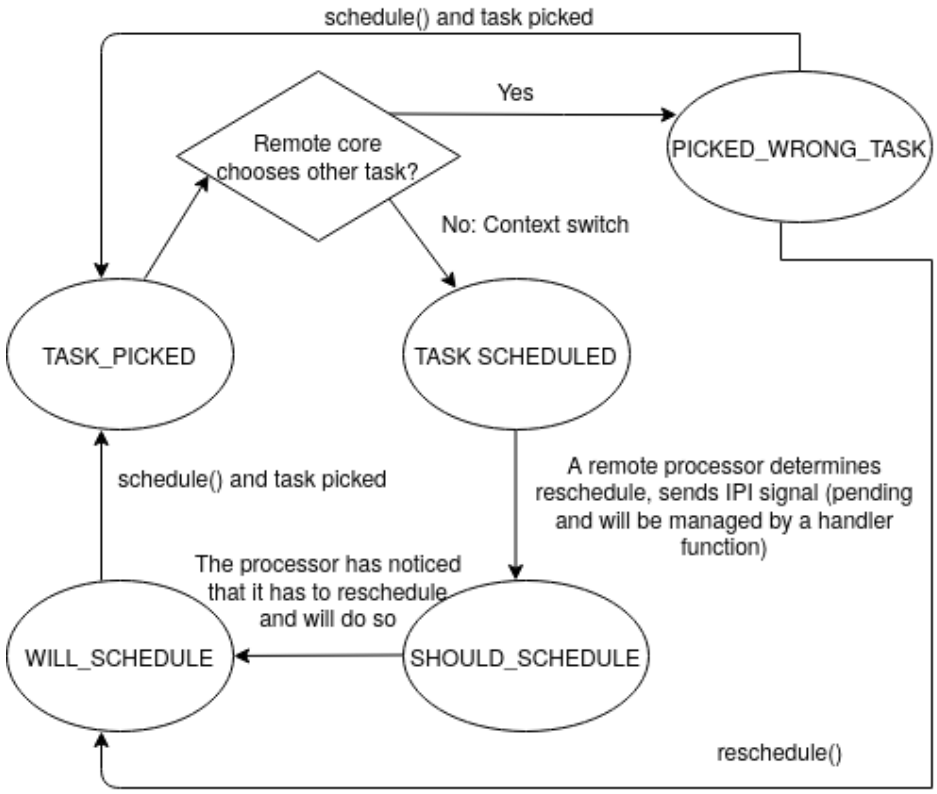


Figure 4.2: Preemptions state machine

With this state machine (Figure 4.2), the Linux kernel can check if the current task is running or if a task preemption is being carried out at the moment. All of this has to be considered just in case the kernel is configured as preemptive on the compilation stage.

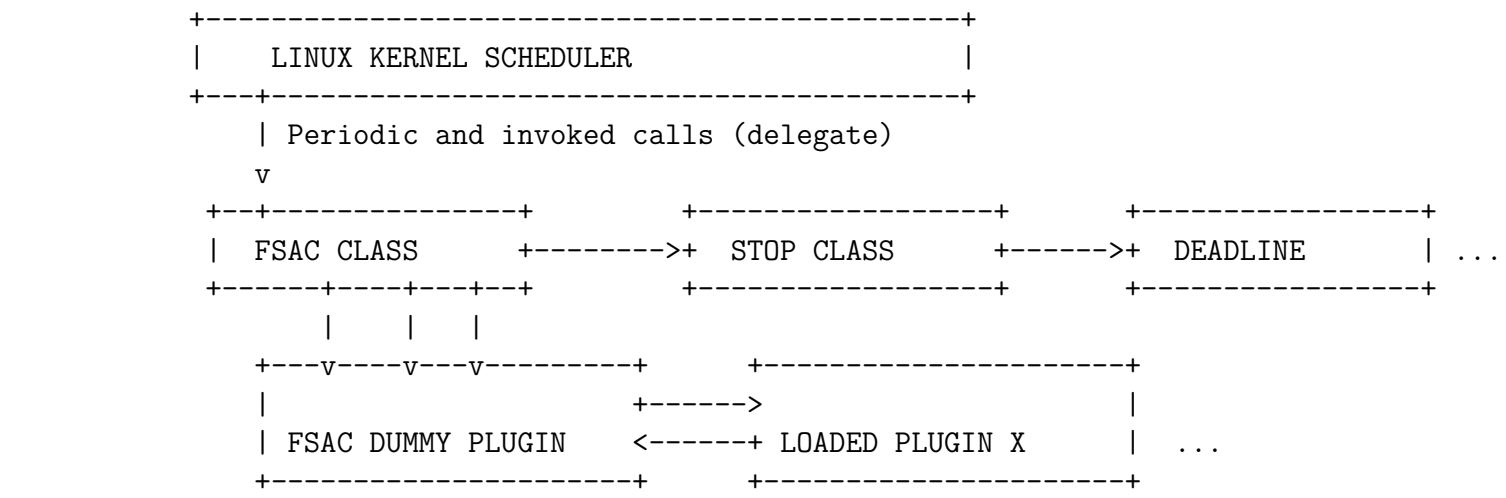
4.4 Linux kernel internal modifications

Now that the design of the API has been covered, we need to perform the necessary modifications of the Linux kernel itself so that the API can be integrated. Therefore, instead of creating new files or directories, most of the work covered in this section will be related with amendments inside the Linux kernel scheduler itself.

The most straightforward task is creating a **generic FSAC scheduling class** that will be added into the linked list of possible scheduling classes of the Linux kernel, covered in Chapter

2. This generic algorithm will be able to dynamically call and delegate on whatever uploaded plugin is at that time activated, using pointers to functions as dynamic hooks.

To avoid runtime calls to null pointers, a "dummy" scheduling plugin *FSAC* should be created, to count with a default active plugin. Needless to say, this plugin will not have any other purpose or use other than avoid null pointers and the plugin will only introduce overhead to the current scheduler if activated. This plugin's code could, nevertheless, be reviewed by developers searching for a source of information to start creating new scheduling plugins. If this new class is developed, after booting and having loaded a new scheduling algorithm, the system should resemble the following diagram:



The scheduling class *SCHED_FSAC* needs to be included, and the file `include/uapi/linux/sched.h` needs to be updated accordingly:

```
#define SCHED_FSAC              7
```

In addition, it is necessary to make *SCHED_FSAC* a valid scheduling plugin in eyes of the kernel, so we have to modify file `/kernel/sched/sched.h`. In addition, the scheduling class FSAC can be included at the first position of the list of scheduling classes, as shown in the chapter devoted to LITMUS-RT. The next thing to do would be to develop this new generic class that will be called by the scheduler's core. As explained before, this class will do the strictly necessary things that every scheduling algorithm should do and delegate the rest in the active plugin. The FSAC class is located in `kernel/sched/fsac_class.c` and the most important aspects of its code are discussed below.

Function `pick_next_task_fsac()` below is in charge of picking the next task to be scheduled. It locks the core run-queue, and calls the FSAC scheduling function of the active plugin. Significantly, we only move the previous task to the waiting threads run-queue after checking there is other task available. The other Linux scheduling classes do this before the scheduling, when they have checked there are queued tasks. As FSAC delegates in plugins, there is no easy way to tell and hence it is easier to check backwards if some other task was found.

```
static struct task_struct *pick_next_task_fsac(struct rq *rq,
    struct task_struct *prev, struct pin_cookie cookie){
```

```
    struct task_struct *next;
    lockdep_unpin_lock(&rq->lock,cookie);
```

The invocation of the current plugin scheduling function could be improved for real-time situations, but it will be enough for now. As we can observe, `put_prev_task()` is called afterwards.

```
    next = fsac_schedule(rq,prev);
    lockdep_repin_lock(&rq->lock,cookie);
    if (next){
        put_prev_task(rq,prev);
    }
  return next;
}
```

Function `fsac_schedule()` needs to take into account the possible SMP configuration. It also needs to refer to the preemption state machine for updates. The SMP configuration is tricky because FSAC needs to check if the global plugin took the task from a different run-queue. If so, the thread needs to migrate the task to that other core; There is no guarantee that the invoking core is the one this thread's run-queue belongs to. The FSAC scheduling class also needs to avoid a concurrent switch race (this could deadlock in the case of cross or circular migrations). In other words, the stack of the next task should not be in used. It will wait until this is fixed, should the scenario occur. After this, FSAC scheduling class needs to make sure the task has not become invalid. If that happens, a rescheduling is triggered. The field used to keep track of the stack from the next task can be updated. This is done with the *fsac_param*, an extra field we have added to the task_struct.

Finally, the scheduling class is added into the linked-list at the first position, and all the function pointers are defined. This is a very important piece of code as defines all the possible interactions that the Linux kernel scheduler can have with the FSAC class and, as a consequence, with its plugins.

```
const struct sched_class fsac_sched_class = {
    .next               = &stop_sched_class,
    .enqueue_task       = enqueue_task_fsac,
    .dequeue_task       = dequeue_task_fsac,
    .yield_task         = yield_task_fsac,
    .pick_next_task     = pick_next_task_fsac,
    .check_preempt_curr = check_preempt_curr_fsac,
    .put_prev_task      = put_prev_task_fsac,
    .task_tick          = task_tick_fsac,
    .switched_to        = switched_to_fsac,
    .prio_changed       = prio_changed_fsac,
    .get_rr_interval    = get_rr_interval_fsac,
    .set_curr_task      = set_curr_task_fsac,
#ifdef CONFIG_SMP
    .select_task_rq     = select_task_rq_fsac,
```

```
#endif
    .update_curr        = update_curr_fsac,
};
```

Another important part of the code is the one related with the plugin switch. This is, when the function `switch_sched_plugin()` is called using a system-call write into one of the special files created for the API in the previous section. This is a pretty complex problem as the plugin could be running in one of the cores. Furthermore, some real-time plugin could be in the middle of its execution. We deal with these scenarios with `/fsac/fsac.c`. Once the adequate precautions have been taken, all the CPUs will receive a callback to the function devoted to switch the plugin.

In addition, it is also important to **prevent the Linux scheduler from balancing workloads** if the scheduling class that is being managed is from the class FSAC. Some *if* statements where added at different kernel files for this purpose, specially in `/kernel/sched/core.c`, but we will not show them here. Subsequently a new file `fsac_param.h` was added at `/include/fsac` with the most basic version of this extra variable. The FSAC scheduling class uses field `stack_in_use` to check if the task's stack is currently in use and avoid deadlocks. It is updated by the FSAC core. Field *present* is true if the task can be scheduled. It is used in the FSAC class function `enqueue_task_fsac()`, for example. Two fields to keep track of the time are included, to help real-time algorithms. Both are initialized at `reinit_fsac_state()`.

```
typedef unsigned long long fsac_time;
struct fsac_param {
    volatile int stack_in_use;
    unsigned int present:1;
    fsac_time last_suspension;
    fsac_time last_tick;
    unsigned int kernel_np; /* As in Non-Preemptive */
};
```

It is utterly important that the FSAC kernel framework does not step into null pointers when the system is running (runtime). To avoid this, when no scheduling plugin has been developed the "dummy" FSAC plugin will be the one hooked so that the FSAC class can invoke something when asked to. This dummy plugin will directly reject any given task. We will now briefly describe its implementation.

The default dummy plugin is implemented in file `/fsac/sched_plugin.c`. We first have the declarations of all the functions the plugins is expected to implement by default.

```
/* (1) Dummy plugin functions */
static long fsac_dummy_activate_plugin(void){ return 0;}
(...)
static void fsac_dummy_task_block(struct task_struct *task){}
static void fsac_dummy_task_exit(struct task_struct *task){}
static ssize_t fsac_dummy_read(char *buf) { return 0;}
```

And we add these functions in to the structure arranged for scheduling algorithms plugins in

the previous section.

```
struct fsac_plugin fsac_sched_plugin = {
        .plugin_name = "FSAC",
        (...)
        .plugin_read = fsac_dummy_read,
};
```

The current plugin at boot time will be the default dummy plugin.

```
struct fsac_plugin *fsac = &fsac_sched_plugin;
```

This same file `/fsac/sched_plugin.c` could be in charge of managing a linked list of the registered plugins, using the previously explained wrappers to ease the linked lists administration. When the register function is invoked, **FSAC will check that all the required functions exist**, to avoid null pointers, and also perform some other security checking. The code shown below is one of the most important components of the FSAC kernel as makes the dynamic management and registration of scheduling algorithms possible. Operator '##' is used to concatenate arguments in the macro.

```
#define CHECK(func) { \
   if(!plugin->func) \
        plugin->func = fsac_dummy_ ## func; }
```

If we want to implement a registering function we need to make sure that the plugin didn't exist before, so we also implement a finding function that we export.

```
struct fsac_plugin* find_sched_plugin(const char* name) {
    (...)
    raw_spin_lock(&proc_plugins_lock);
    plugin = fsac_find_node(0,name,sched_plugins);
    (...)
}
EXPORT_SYMBOL(find_sched_plugin);
```

The other main functionality of this file is managing the list of registered plugins. In order to avoid duplication and saving memory, we decided to use the same linked list as for the `/proc` management (this is, `/fsac/fsac_proc.c`) and the registering/unregistering of plugins.

Inside the Linux kernel, there are several functions that take into account that the scheduling class to which, the thread that is currently running belongs, could be real-time. This is done in order to **avoid delays in their execution**, or even live-locks. If a developer intends to use this framework (FSAC) to develop a real-time scheduling algorithm, all these precautions should be taken as well. Hence, some modifications are required, starting by providing a way for the developer to specify if he/she wants to develop a real-time scheduling algorithm, where every nanosecond counts. In order to do so, a new field can be added into the structure *sched_plugin*, from the API discussed in the previous section. When registered, the framework will check if the value of the field is set to one (real-time), and the default value will be zero (non real-time).

Several files from the Linux kernel have to be modified too, so that the real-time FSAC plugins could receive the same treatment as the real-time class' tasks. Table 4.1 shows the files that had to be modified along with the reason why:

Table 4.1: Kernel files modified to support real-time plugins in FSAC

FILE	MODIFICATION
`fs/select.c`	Fixed required to avoid delays if the FSAC plugin is real-time.
`kernel/locking/mutex.c`	Fixed required to avoid live-lock if the FSAC plugin is real-time.
`kernel/locking/rwsem-xadd.c`	Same idea as above but with semaphore mechanisms.
`kernel/time/hrtimer.c`	Same idea as above but for the kernel timer.
`mm/page-writeback.c`	FSAC RT tasks should get special treatment when it comes to the writing-back of dirty pages at the address space.
`mm/page_alloc.c`	Special amendments at page allocation when the FSAC plugin is intended to be real-time.

Chapter 5

PMCSched: A PMCTrack-based scheduling framework

The design and implementation of the PMCTrack-based [7] scheduling framework are described in this chapter. One of the main reasons we chose PMCTrack was that it has support for hardware events monitoring, making use of the advanced Intel RDT technologies. We will introduce them in the following section. In addition, we discuss the API we have included to help developers prototype scheduling algorithms. In the last section, a coarse-grained Round Robin scheduling algorithm is developed with PMCSched to illustrate how to create a new algorithm.

5.1 PMCTrack support for Intel RDT technologies

As previously explained, shared resource contention poses a serious problem on current multi-core systems. Applications executed simultaneously can experience performance degradation due to an uneven distribution of shared memory resources. This constitutes an unfair situation [2], [14], [17], but is a consequence of the very nature of memory hierarchy. That is, several components such as the last-level cache (LLC) or the DRAM controller are typically shared. As the contention occurs naturally, it is imperative to find ways to improve scheduling algorithms so that they take it into account and act in accordance.

Granted, researchers acknowledged this and several memory-cognizant algorithms have been proposed in the last decade [14], [17]. Contention-conscious algorithms take into account the consumed bandwidth and effectively monitor cache utilization for each of the threads. Depending on the application's cache-related behavior, threads may be classified to different categories. Those categories are later used to parametrize the scheduler and help the scheduler decide. One of the most broadly adopted classification labels threads as *streaming* (working set size bigger than LLC size and/or has a low-reuse access pattern to the cache), *cache intensive* (working set size does not fit in the private cache levels, intensive bandwidth usage and may not reuse cache heavily) or *core fitting* (working set size fits within private cache levels).

Nevertheless, even with all the proposed contention-conscious algorithms, the same problem as in previous subsection arises. Developing those in the kernel is an arduous task. For that reason, some researchers opted to do so via user-space prototypes. Nonetheless, the ease of development comes at the expense of an **important overhead** [6]. As a matter of fact, many low-level events such as the number of cache misses simply cannot be monitored from there.

We wanted to make it easier for researchers that aim to implement novel contention-aware scheduling algorithms. Thereby, we needed a kernel version that included kernel-level scheduling support for Intel RDT cache and bandwith allocation and monitoring technologies. Essentially, what we had in mind was an API and a framework -just like with FSAC- but that also featured tools for building contention-aware schedulers.

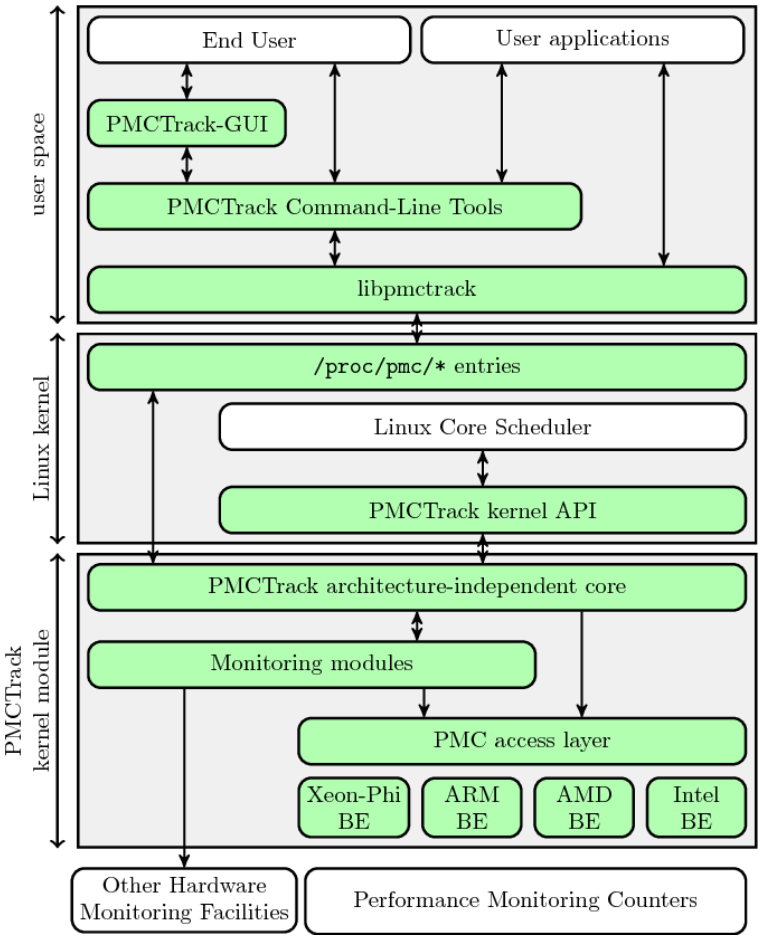


Figure 5.1: PMCTrack architecture [27]

Instead of using the FSAC kernel, which is a very basic framework in its most primitive version, **adding this functionality to PMCTrack seemed the most appropriate choice**. The open-source PMCTrack project [7], [26] started as an OS-oriented performance monitoring tool for GNU/Linux, but has grown into something way more ambitious, with countless interesting utilities for system developers. One of these is the ability to add monitoring tools as part of a loadable kernel module, easing programming and debugging processes. Hence, including a new flexible framework monitoring module into PMCTrack that allowed prototyping of new

cache-cognizant scheduling algorithms (as *plugins*), refereed to as PMCSched, appeared to be the most promising approach.

In order to create the PMCSched framework we leveraged the monitoring modules' infrastructure of PMCTrack. Monitoring modules are platform-specific components that enable to extend the utilities PMCTrack has to offer. Figure 5.1 illustrates the interaction of the PMCTrack kernel module, which implements most of the functionality, and the patched Linux kernel. The PMCTrack module retrieves important scheduling events (thread creation, context switches…) and gathers performance counter data from each thread. To be able to feed with information from the scheduling classes, the PMCTrack kernel API is included inside the kernel's core. This requires some minor modifications of the kernel scheduler, in the form of a patch. To fulfill the project goals, we enlarged that patch to be able to use more abstractions that the scheduler provides to the scheduling classes.

Some of the most relevant functions provided with the PMCTrack's kernel module interface are the following:

```
typedef struct pmc_ops{
/* Invoked when a thread is created */
void* (*pmcs_alloc_per_thread_data)(unsigned long, struct task_struct*);
/* A thread leaves the CPU */
void (*pmcs_save_callback)(void*, int);
/* A thread enters the CPU */
void (*pmcs_restore_callback)(void*, int);
/* A process invokes exec() */
void (*pmcs_exec_thread)(struct task_struct*);
/* A thread exits the system */
void (*pmcs_exit_thread)(struct task_struct*);
(...)
} pmc_ops_t;
```

5.2 Building the PMCSched API

One of the first steps in the development stage was to prepare PMCTrack for the dynamic switching of scheduling algorithms. Using a similar scheme as the one employed for FSAC plugins, we include a structure *scheduler_type* with the minimal set of callback functions that a scheduling algorithm in PMCSched would need.

```
typedef struct scheduler_type {
        sched_policy_mm policy;
        struct list_head link_schedulers;
        sched_kthread_periodic_t sched_kthread_periodic;
        on_active_thread_t       on_active_thread;
        on_inactive_thread_t     on_inactive_thread;
        on_exit_thread_t         on_exit_thread;
} scheduler_skeleton_t;
```

To define the policy of the scheduling plugin and allow the insertion and deletion of the algorithm to linked lists, we respectively include fields `policy` and `link_schedulers`. The next fields are functions. These include a periodic invocation to the scheduling algorithm, and functions to deal with threads that get activated (a thread becomes runnable), deactivated (the thread is stopped or blocked) or directly exit the system.

In addition, we have to include structures to manage each of the threads created with PMCTrack, the applications associated to them, and the general view from the system. This structures became more complex as new things needed to be added, hence we will just explain here the main purpose of each of the fields in the final structures.

First, we have an structure for the application, which can be constituted by one or more threads (making it single-threaded or multithreaded). This includes:

```
typedef struct app {
        sized_list_t app_active_threads;
        sized_list_t app_stopped_threads;
        struct list_head link_active_apps;
        struct list_head link_stopped_apps;
        atomic_t ref_counter;
        int state;
} app_t;
```

The field `app_active_threads` is a special abstract data type (ADT) that manages a linked list of active threads belonging to that application. Same logic goes for the stopped applications with `link_stopped_apps`, which have no running threads in the system. The field `link_active_apps` is used to manage a list of the available active applications. The inverse is applies for `link_stopped_apps`. Needless to say, the application should not be present at both of these lists simultaneously. Finally, the field `ref_counter` is used to keep track of the number of references to that application, and being able to perform operations emulating the job of a garbage collector. The state of an application can be NO_QUEUE, STOP_QUEUE and ACTIVE_QUEUE, depending on the position of the application within the aforementioned lists.

The next structure we now proceed to discuss represents the **global state of the threads**, required for the scheduler. It is important to remark the `lock`field spin-lock, acquired before scheduling decisions, as well as the linked list of available schedulers, which has to be initialized with special care. We include linked lists for:

```
struct {
        sized_list_t active_threads;
        sized_list_t stopped_threads;
        sized_list_t active_apps;
        sized_list_t stopped_apps;
        sized_list_t schedulers;
        sized_list_t all_threads;
        spinlock_t lock;
} sched_prot_gbl;
```

The active or stopped threads on the system, which will always be part of their corresponding applications are the first and second fields respectively. The lists of active and stopped applications are `active_apps` and `stopped_apps`. If an application is stopped it cannot have any thread in the active threads list.Finally, we include a linked list for all the developed schedulers, for all the threads in the system (stopped or not) and a spin-lock to allow synchronization.

The most important structure from the point of view of the algorithms is the *sched_prot_thread_data_t*. Intuitively, the structure contains the necessary per-thread fields required by any contention-aware scheduling algorithm to function. We also use it to describe each thread that PMCSched should take care of. The most relevant fields are:

```
typedef struct {
        metric_experiment_set_t metric_set; /* Two core types */
        unsigned int runnable;
        pmon_prof_t* prof;          /* Backwards pointer                    */
        app_t* app;                 /* Application this thread belongs to */
        struct list_head link_active_threads_gbl;
        struct list_head link_active_threads_apps;
        struct list_head link_stopped_threads_gbl;
        struct list_head link_stopped_threads_apps;
        int security_id;
        cpumask_t* mask;
        struct list_head migration_links; /* For CPU masking */
        struct list_head signal_links; /* To signal threads */
        int state;
} sched_prot_thread_data_t;
```

The field `app` is needed to relate the thread with its application and *prof* to retrieve the pid of the task (it is a special field added to the structure *task_struct*). We also include global lists for the active threads and applications, and similarly for the stopped threads and applications.

The `list_head _migration_list` is of big importance. It is the mechanism we give the scheduling plugin to **bind the threads to specific cores** (using the mask field). The plugin itself cannot bind threads to CPUs and has to delegate on PMCSched. That is because changing the CPU-affinity mask of a thread (`set_cpus_allowed_ptr()_`) involves calling a blocking function and so it cannot be invoked in interrupt context, where the plugin will be at due to the way PMCTrack implement callbacks on the Fair class context switches. Using SoftIRQs or Tasklets would not solve the problem. Therefore, we need to **use a kernel thread** (Workqueue kworker/X), that will be executed in process context, to perform the periodic scheduling. Unfortunately, this has some drawbacks in terms of performance.

The `signal_links` list follows a similar logic as the migration list. If the plugin wants PMCSched to send a specific signal to a thread, it only needs to add it to the list. The scheduler will later check the list (we will explain how later) and send the corresponding signals. The scheduler will **infer the signal** that needs to be sent according to the state of the threads.

Finally, field `state` represents the situation of the thread according to PMCTrack, depending on the signals that are requested and the queue it is on. In the next section, we will explain in more detail the importance of keeping a precise record of the situation of the threads within the scheduler. The following Table 5.1 summarizes the possible states:

Table 5.1: Possible states for threads in PMCSched

Table	Description	Value
NO_QUEUE	The plugin has not assigned a queue yet	0x0
STOP_QUEUE	The thread is in the stopped threads queue	0x1
ACTIVE_QUEUE	The thread is in the active threads queue	0x2
STOP_PENDING	The plugin wants to stop this thread	0x3
ACTIVE_PENDING	The plugin wants to wake this thread	0x4
KILL_PENDING	The plugin wants to kill this thread	0x5
TASK_KILLED	The thread was killed by someone	0x6
REACTIVATED	State reserved for special algorithms	0x7

Other crucial aspect of the PMCSched API has to do with the developer's interaction with the plugin. PMCTrack already has support to easily gather data from performance monitoring counters (PMCs), so we just need to modify the read/write functions implemented in communication with the `/proc` file system (just like FSAC) to allow the **dynamic switching of scheduling plugins**.

The implementation of the reading function is rather straightforward, as we just need to add the current state, which is the active plugin, the value of the verbose option and the available plugins, along with some extra indications. Hence, we will not discuss the code here, but rather just so the output:

```
zildj1an@iron_maiden:~/$ cat /proc/pmc/config
sched_sampling_period = 1000
kernel_buffer_size = 3952 bytes (26 samples)
-----------------------------------------------------------------------
Verbose option for kernel log => ON
The developed schedulers are:
[*] 0 - Dummy default scheduler
[ ] 1 - Coarse-grained Round Robin scheduling policy
[ ] 2 - Co-scheduling algorithm (8) cores
[ ] 3 - Memory-aware algorithm
To change the active scheduler echo 'scheduler <number>' here
and to switch the verbose option (kernel log) echo 'verbose <1/0>'
-----------------------------------------------------------------------
```

On the contrary, the write operation needs to be developed with more care as we will be changing the current plugin at runtime.

```
static int sched_prototype_on_write_config(const char *str, unsigned int len)
```

```
{ (...)
 if (sscanf(line,"scheduler %d",&val)==1 && val<NUM_SCHEDULERS) {
   aux = head_sized_list(&sched_prot_gbl.schedulers);
   for (i= 0; i < NUM_SCHEDULERS && !found && aux != NULL; i++,aux=next){
        next = next_sized_list(&sched_prot_gbl.schedulers,aux);
        if (aux->policy == val){
         found = 1;
         pending_threads = sized_list_length(&sched_prot_gbl.stopped_threads)
                         + sized_list_length(&sched_prot_gbl.active_threads);
```

The plugin should not be changed if there are pending threads that need to be scheduled in the previous plugin. If the developer, aware of the danger, still wants to force the switching, he/she can do so by changing PMCTrack active module, to any other PMCTrack module, and then switching back to PMCSched. This will make PMCSched kill all the pending threads, but it is not the curse of action we recommend, as could lead to unexpected behavior or an unstable system even with the precautions we have taken to make that scenario safe.

```
         if (!pending_threads){
                printk(KERN_ALERT "New scheduler set (%d)...\n", val);
                active_scheduler = aux;
                BUG_ON(!active_scheduler->sched_kthread_periodic);
                BUG_ON(!active_scheduler->on_active_thread);
                BUG_ON(!active_scheduler->on_inactive_thread);
                BUG_ON(!active_scheduler->on_exit_thread);
```

As it can be observed above, we check if the developer has implemented all the required functions, as we do not want to step into null pointers in the future. All the necessary scheduling functions need to be implemented by the developers before, or else the `BUG_ON()` macro will execute something with an invalid opcode and **the CPU will throw an exception**, displaying a long alert message at the kernel buffer log. In addition, all the general system lists need to be reinitialized:

```
                init_sched_prot_gbl();
        }
        else (...) /* An error message */
```

Finally, the developer can also update the verbose option via the `/proc` interface, and this will affect the number of displayed messages.

To conclude with the description of the API, we will mention the purpose of other functions that every plugin could use to save some time. For example, the following can be called by the plugin to update the state of the application associated with a thread recently stopped.

```
__attribute__((used)) static void check_inactive_app(struct app *app){
 if (!sized_list_length(&app->app_active_threads)&& app->state!=STOP_QUEUE) {
    if (app->state == ACTIVE_QUEUE)
        switch_queues(&sched_prot_gbl.active_apps,&sched_prot_gbl.stopped_apps,app);
    else
```

```
insert_sized_list_tail(&sched_prot_gbl.stopped_apps,app); (...)
```

To summarize, Figure 5.2 below illustrates the most important components of the infrastructure available for any new scheduling plugin, whose features will be used in the next section to develop the module internals. Along with this API we can also count with all the handy tools and resources that were already available in PMCTrack.

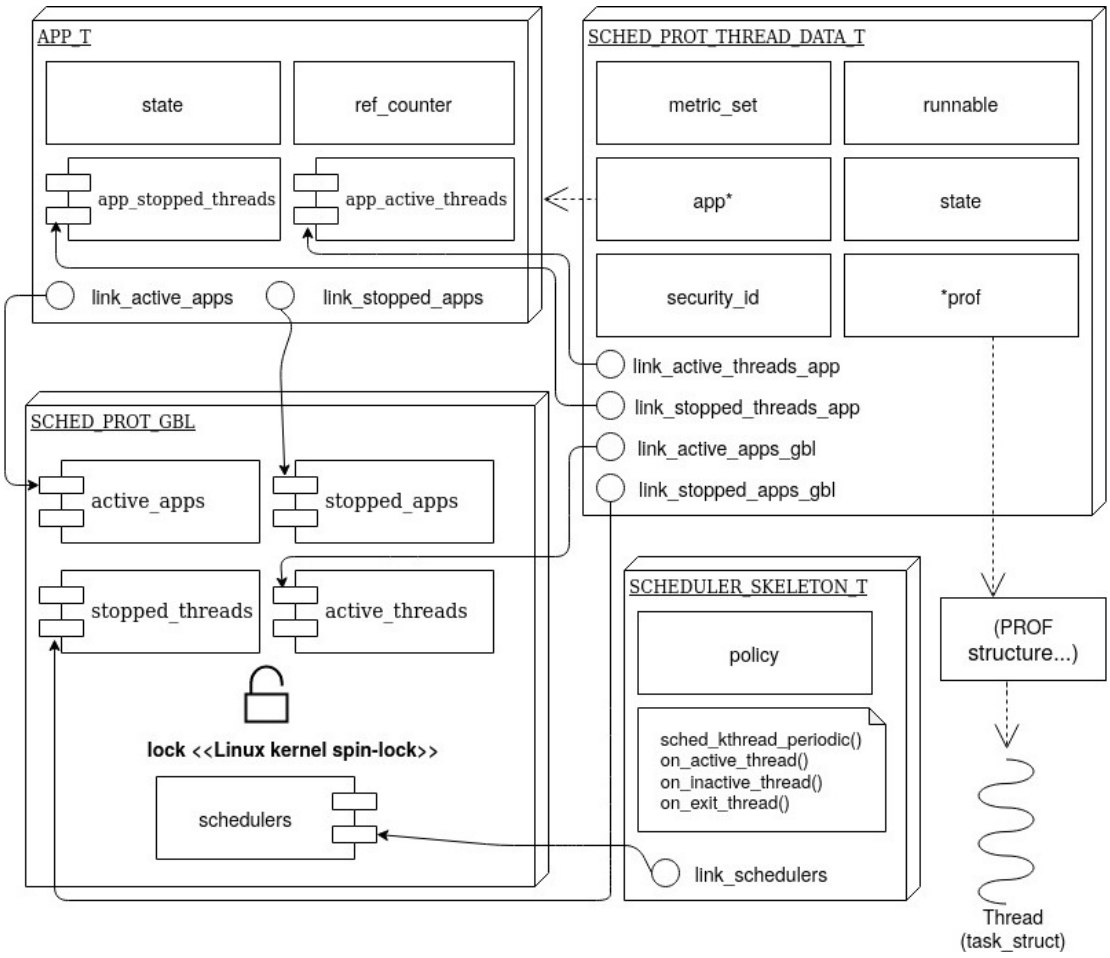


Figure 5.2: PMCTrack scheduler API main structures

5.3 Design and implementation of PMCSched

PMCSched should take care of -following the same pattern as for the FSAC scheduling class- all the necessary generic background work that any scheduling algorithm must do. This way, the developer's workload can be highly reduced and some abstractions can be used to develop new algorithms and ease the process. In this section we provide a high-level description of the internal design of PMCSched, corresponding to file `sched_prototype_mm.c` at PMCTrack's source code.

The basic idea of the PMCSched is to **delegate specific details of the algorithm** to the active scheduling plugin from among the available algorithms. For instance, when a task exits,

it should mainly delegate:

```
spin_lock_irqsave(&sched_prot_gbl.lock,flags);
active_scheduler->on_exit_thread(t);
spin_unlock_irqrestore(&sched_prot_gbl.lock,flags);
```

In most cases, apart from making sure that there are no pending signals (with a kernel semaphore) and that the task was creating using the tools provided by PMCTrack, the generic scheduler will simply delegate on the plugin to handle the task with a new state according to the details of the current algorithm. As only minor checks are done at these functions, we will not describe them here nor show their implementation.

There is, nonetheless, a notable design difference between the FSAC scheduling class and PMCSched that is worth discussing. That is that PMSched allow us to create new scheduling algorithms within a kernel module. As the reader might remember, FSAC was a direct modification of the Linux scheduler, and everything is developed inside the kernel itself.

Therefore, even-though some design ideas can be reused, most of the things need to be done differently. For instance, the module will need to use **signals to force the thread to sleep** or wake up (these signals are required to perform scheduling decisions in PMCSched), and **CPU affinity masks** to bind processes to specific cores, in a non-interruptible context (see Figure 5.3) .

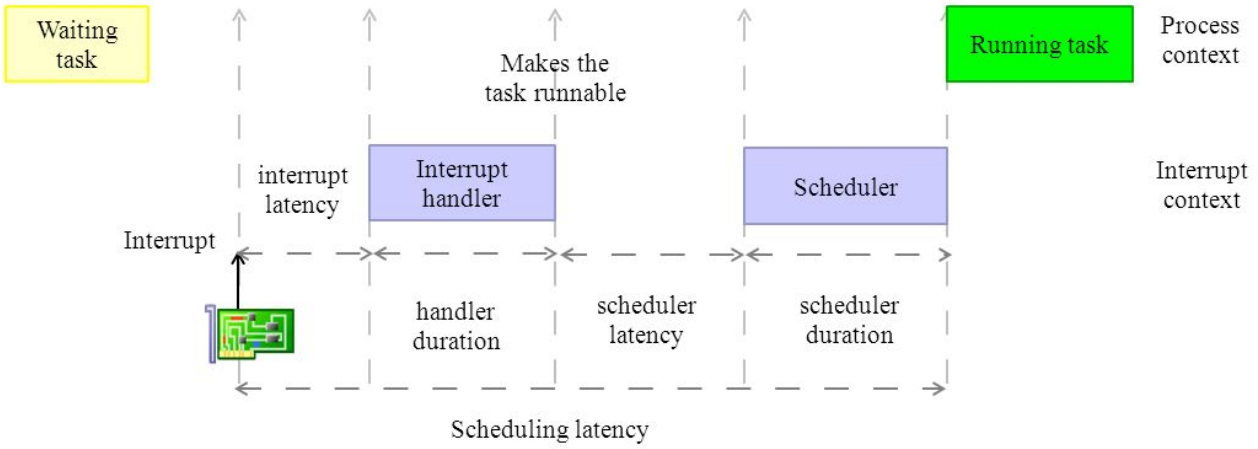


Figure 5.3: Interrupt Context (Authorship Thomas Petazzoni)

This approach posed several challenges, mainly related to the SMP design of the Linux kernel. For example, as the previous section advanced, **the scheduler cannot delegate on the plugin to bind threads to cores** as this is a blocking call, and as such it can lead to deadlocks. If no interrupts can be handled the core would suffer a RCU (Read-Copy-Update) stall.

The same occurs when **sending signals**, as the plugin cannot directly wake-up or put to sleep a thread because interrupts are disabled before invoking the plugins and, while we wait for the thread to change its state, it would never be able to do so as the signal cannot be

handled with a callback. In both cases (signal sending and core-to-thread assignments) we opted for passing a list to the plugin that can populate with threads to be stopped/waken-up. The scheduler will then infer the signal that should be sent based on the state of the task (see Table 5.1).

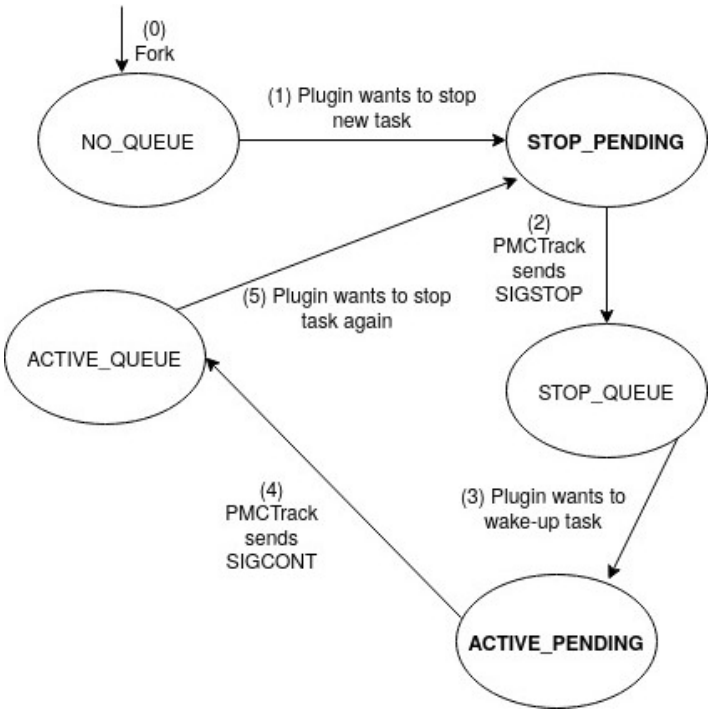


Figure 5.4: PMCSched thread states

The plugin should only update the states of the thread to pending cases, such as ACTIVE_PENDING (see Figure 5.4), in order to keep a realistic record log of the system's state. Even though the thread is in the active queue, it is not yet running and *many* things could go wrong before the wake-up signal is delivered and successfully handled.

PMCSched includes a **periodic kernel thread** to invoke the plugin periodic function to make scheduling decisions and handle pending signals, profilings and thread-to-core assignments. PMCSched passes to the plugin the lists for the bit mask and the signal-sending as parameters.

The periodic kernel thread is initialized when the module is loaded:

```
kthread = kthread_run(sched_kthread_periodic, NULL, "scheduler_kthread");
```

Later when invoked, it will enter in an endless loop with small periods of sleep. HZ is denoted in ticks, and we have set it to HZ/2, which is approximately 500ms.

We also set real-time priority to make the kernel thread kick in faster.

```
sched_setscheduler(current, SCHED_RR, &param);
```

Finally, we periodically call the scheduling function unless the monitoring module is disabled. The *disabling* mode is set in case we detect the developer is forcedly disabling the module with pending threads, in which case we should not invoke the plugin.

```
    while (!kthread_should_stop()) {
        set_current_state(TASK_INTERRUPTIBLE);
        schedule_timeout(DELAY_KTHREAD); /* Sleep */
        if (!disabling_mode) __periodic_scheduling();
        else check_pending_signals();
    }
```

When invoked, the periodic scheduler should prepare the call to the plugin, i.e. initialize the lists for signal-sending, check if there are no pending threads by increasing a kernel semaphore, lock and disable interrupts. When all of that is done, we can finally delegate on the currently active plugin:

```
active_scheduler->sched_kthread_periodic(&migration_list,&pending_signals_list);
spin_unlock_irqrestore(&sched_prot_gbl.lock,flags); /* Release lock */
up(&signals_semaphore); /* Release signals semaphore */
```

After the periodic invocation to the plugin, the scheduler will receive an updated list of threads that need to be masked and will do so by retrieving the thread identifier embedded in the structure just like described in previous section (see Figure 5.2). We first made the scheduler send as a parameter a CPU-mask variable that the plugin could update, but with this approach (a list), different threads can be assigned to different cores in the same invocation to the function.

If there are threads to be migrated, we iterate over the list filled by the plugin.

```
if (sized_list_length(&migration_list)){
    elem = head_sized_list(&migration_list);
    for (task = 0 ; task < sized_list_length(&migration_list) && elem != NULL;
        task++){
```

We retrieve the task structure from each of our threads and mask them using the proper kernel abstractions.

```
        if (set_cpus_allowed_ptr(elem->prof->this_tsk, elem->mask) < 0){
            printk(KERN_ALERT "%s: set_cpus_allowed_ptr failed.\n",__func__);
            break;
        }
        elem = next_sized_list(&migration_list,elem);
    }
}
```

We added two more functions to facilitate the signal delivery. The first one, `check_and_send()`, will receive as parameter a linked list of threads that need to be signaled and, after checking the signals semaphore and that the list is not empty, *infer* the signal that should be sent based on the state of each element of the list.

```
    switch(elem->state){
        case ACTIVE_PENDING:
            __send_signal(SIGCONT,elem->prof->this_tsk);
```

```
            elem->state = ACTIVE_QUEUE;
        break;
        case STOP_PENDING:
            __send_signal(SIGSTOP,elem->prof->this_tsk);
            elem->state = STOP_QUEUE;
```

Notably, we decided that the state for when a thread is going to be killed should be updated *before* sending the signal. That is not arbitrary, but:

1. Helps differencing between a SIGKILL sent by the plugin and a developer that has sent a kill signal, perhaps using `$kill -s 9`. Other prototyping solutions developed at user-space do not take into account such external influences that could easily lead to *kernel panics*. For instance, if the thread is killed and then one of the plugins attempts to access or modify a value from the structure *task_struct* that would lead to a system crash.

2. It is conceptually safe to assume that the state of the thread is going to be killed, as SIGKILL signals cannot be handed by the thread, and IRQs are enabled.

```
        case KILL_PENDING:
                    elem->state = TASK_KILLED;
                   __send_signal(SIGKILL,elem->prof->this_tsk);
                break;
```

For each of these states, the second function we implemented is called, `__send_signal()` which will be the one effectively sending the signal to the thread. This function is in charge of sending the signal to the specific thread and does not update its state.

```
void __send_signal(int sig_num, struct task_struct* p){
   struct siginfo info;
   unsigned long previous_state = p->state;
   int ret = 0;
```

The memory for the structure *siginfo*, used in the kernel for signal-delivery, is allocated and the variables are initialized:

```
   memset(&info, 0, sizeof(struct siginfo));
   info.si_signo = sig_num;
   info.si_code = 0;
   info.si_int = 1234;
```

Reached this point, disabled interrupts could potentially freeze the whole system. We need to check that this is not the case. After that, we can send the signal.

```
   BUG_ON(irqs_disabled());
   if ((ret = send_sig_info(sig_num, &info,  p)) < 0) {
           printk(KERN_ALERT "Error sending signal to PID %d\n",p->pid);
           return;
   }
```

Lastly, we need to make sure that the signals have been properly captured by their corresponding threads. In the case of the SIGSTOP signal, by using an exported atomic kernel function `wait_task_inactive()` and in the case of a SIGCONT signal, idling the core until the signal handling has taken place.

```
switch(sig_num){
 case SIGSTOP:
     wait_task_inactive(p, p->state);
     break;
 case SIGCONT:
     while (previous_state != TASK_RUNNING && previous_state != TASK_WAKING &&
         p->state == previous_state){
          cpu_relax();
     }
     break;
 (...)
}
```

On the other hand, we can safely assume that the delivery of the killing signal will work, as no thread can avoid it. Nevertheless, it might delay a few instants.

```
case SIGKILL:
    while (p->state != TASK_DEAD){
        cpu_relax();
    }
    break;
```

5.4 A coarse-grained Round Robin algorithm in PMCSched

In order to test both the usability of the API and the robustness of PMCSched, we first developed a coarse-grained Round Robin algorithm as proof of concept. Hence, this subsection, apart from a brief explanation of the logic behind this plugins, can also serve as a tutorial for anyone interested in developing a new scheduling algorithm using PMCSched's plugins abstractions.

First, we need to define the plugin. We add it to the header file `sched_prototype_mm.h` so that PMCSched can manage it.

```
#define SCHED_RR_MM  1
extern const struct scheduler_type rr;
```

Subsequently, we create a file `rr_plugin` to start developing the scheduling plugin. In the structure, we can also include a description of the algorithm that will be displayed in `/proc` using the read function described before.

```
struct scheduler_type rr = {
```

```
    .policy = SCHED_RR_MM,
    .description = "This is our coarse-grained RR algorithm",
    .sched_kthread_periodic = sched_kthread_periodic_rr,
    .on_active_thread = on_active_thread_rr,
    .on_inactive_thread = on_inactive_thread_rr,
    .on_exit_thread =  on_exit_thread_rr,
};
#endif
```

Thereafter, we need to implement each function of the scheduling plugin interface. First, for any task that becomes active for the first time, we would like to stop it and move it to the stop queue. It should only be moved to the active threads queue by the periodic scheduling, following the coarse-grained Round Robin approach.

If the task that just became active was already in the global active threads queue, nothing has to be done. Otherwise, if the task that just became active was in the global stopped threads queue or if it was in no queue whatsoever, it will have to wait for its turn to be moved into the active threads and so it should be stopped. **The plugin cannot assume the thread is not in a signal pending state**, and we only really care about if the task should be stopped or it is new. This function will be called in two scenarios: The task becomes active when started, or the task is moved from the stopped queue, with the precondition of being runnable. As shown in the previous section, `check_inactive_app()` is used to check if it became an inactive application and is updated consequently. We start checking if we can manipulate it:

```
static void on_active_thread_rr (sched_prot_thread_data_t* t, sized_list_t*
    signal_list) {
        if (t->state != ACTIVE_QUEUE && !signal_pending_th(t)) {
            t->runnable = 1;
```

If the condition holds true, we have to add the structure to the correspondent per-application and global lists and update the state accordingly. First, we can deal with the scenario in which it is in any queue, because it is a newly created thread.

```
            if (t->state == NO_QUEUE){
                insert_sized_list_tail(&t->app->app_stopped_threads,t);
                insert_sized_list_tail(&sched_prot_gbl.stopped_threads,t);
                t->state = STOP_PENDING;
            }
```

Finally, we add the thread to the linked list of processes whose signal have to be checked by PMCSched, as we want it to be stopped.

```
        insert_sized_list_tail(signal_list,t); /* For signal-sending */
        check_inactive_app(t->app);
```

Again, it is important to remember that we cannot stop the thread here directly (sending SIGSTOP), as we are working in a context change state with interrupts disabled.

The logic for when a thread becomes inactive (`on_inactive_thread_rr()_`) is very similar so

we will not discuss it here. We would instead like to review the periodic scheduling function of the coarse-grained Round Robin plugin, that provides more interesting insights for scheduling algorithms developers with PMCSched. As the reader might remember, this function is called approximately every ~500 ms by a kernel thread initialized by PMCSched. It receives lists to fill with signals that would like to be sent or threads that have to be masked to specific cores.

For this coarse-grained Round Robin plugin, we will make the rescheduling of the selected threads to take place every two iterations, or approximately `2 * DELAY_KTHREAD = 2*(HZ/2) i.e. each HZ = ~1 seconds`. We will make the first thread (as it is a RR) of the list of stopped threads start running with signals. Hence, it will be moved at the tail of the active threads. Since at least one of its threads is running, the application is consequently moved into the active applications list. Similarly, the head of the active threads is moved into the tail of the stopped threads and stopped (signal). The application it belongs to is therefore moved into the stopped application list if that was the last running thread. In addition, CPU mask is set, core 1 if "schedulings" is a multiple of 5. Else core 0 (we check that system has at least two logical cores). This will happen each `5 * DELAY_KTHREAD = 5*(HZ/2) i.e. each ~2.5 seconds` (again, this value might differ slightly depending on the system).

We start declaring the correspondent variables and checking that there are stopped threads on the system. If there are not, there would be no need to reschedule.

```
void sched_kthread_periodic_rr (sized_list_t* migration_list,
    sized_list_t* signal_list)  {
    static int task, schedulings = 1, on_cpu, cpu_prev=-1,cpu;
    sched_prot_thread_data_t *activated, *stopped, *elem;
    if (likely(sized_list_length(&sched_prot_gbl.stopped_threads))){
        elem = head_sized_list(&sched_prot_gbl.stopped_threads);
```

If there are stopped threads, we will iterate until we find the first runnable thread. Being or not runnable is a property of the thread itself and the plugin should never update this unless the thread itself becomes active.

```
    activated = NULL;
    for (task = 0 ; activated == NULL && task < sized_list_length(
        &sched_prot_gbl.stopped_threads) && elem != NULL; task++){
            if (elem->runnable && !signal_pending_th(elem)){
                activated = elem;
                break;
            }
            elem = next_sized_list(&sched_prot_gbl.stopped_threads,elem);
    }
```

If the scheduler finds some stopped thread runnable, the thread is moved from the queue of stopped threads to the tail of the active threads, following the coarse-grained Round Robin algorithm specification. Its state is updated to ACTIVE_PENDING and it is added to the list of threads with pending threads. When the function returns **PMCSched will infer the signal** that has to be sent as explained in the previous section 5.3.

```
if (activated){
    switch_queues(&activated->app->app_stopped_threads,
        &activated->app->app_active_threads,activated);
    switch_queues(&sched_prot_gbl.stopped_threads,
        &sched_prot_gbl.active_threads,activated);
    activated->state = ACTIVE_PENDING;
    insert_sized_list_tail(signal_list,activated);
    check_active_app(activated->app);
}
```

Similarly, the thread at the head of the active threads list, if any, will be stopped. Granted, the list of active threads should be bigger than one as we cannot kick the recently added thread. Finally, we mark the application that thread belongs to as inactive if that was its last running thread.

```
if (likely(sized_list_length(&sched_prot_gbl.active_threads) > 1)) {
    stopped = head_sized_list(&sched_prot_gbl.active_threads);
    if (!signal_pending_th(stopped)){
        switch_queues(&activated->app->app_active_threads,&activated->app->
            app_stopped_threads,activated);
        switch_queues(&sched_prot_gbl.active_threads,&sched_prot_gbl.
            stopped_threads,activated);
        stopped->state = STOP_PENDING;
        insert_sized_list_tail(signal_list,stopped);
        check_inactive_app(stopped->app);
    }
}
```

Now that we are done with that, we can take care of the **binding of threads to cores**. Our coarse-grained Round Robin algorithm sets the per-thread affinity mask only if there are at least two cores in the system. For this example we decided to bind the thread to CPU 1 if the invocation was a multiple of five and to the CPU 0 otherwise. We will perform the migration if the invocation number is a multiple of 5, a situation that will occur approximately each ~2.5 seconds.

```
if (num_online_cpus() >= 2){
    cpu = !(schedulings % 5)? 1 : 0; /* core 1 if multiple of 5 */
    if (cpu != cpu_prev && sized_list_length(&sched_prot_gbl.active_threads)){
        elem = head_sized_list(&sched_prot_gbl.active_threads);
        cpumask_set_cpu(cpu, elem->mask);
```

We need an auxiliary migration list as we cannot invoke `sched_setaffinity()` (a blocking function) with the lock acquired. In addition, it is a possible scenario that the thread does not need to be bound if it is **already running on that core** right now, something that we can check that with *task_cpu()*.

```
        for(task = 0 ;task < sized_list_length(&sched_prot_gbl.active_threads)
            && elem != NULL; task++){
```

```
on_cpu = 0;
on_cpu = (task_cpu(elem->prof->this_tsk) == cpu);
```

If it was not already on that core we will trigger a migration:

```
if (!on_cpu && !signal_pending_th(elem))
    insert_sized_list_tail(migration_list,elem);
```

Chapter 6

Contention-aware scheduling algorithm

In this chapter we will discuss the development of two scheduling algorithms using the PMCSched framework: a n-cores co-scheduling algorithm and a contention-awaee scheduling algorithm, that leverages several mechanisms from the first one. A scheduling algorithm is of co-scheduling [19] if it makes scheduling decisions over a set of processes applying some grouping criteria, instead of on a thread-to-thread basis.

6.1 Motivation

Intel recently launched **Intel RDT** [8]–[12], a set of technologies to perform monitoring and partitioning of the L3 cache and adjusting memory bandwidth consumption. With the Intel CMT (Cache-Monitoring Technology) mechanism included on recent Intel processors, the OS can be informed at runtime about the amount of bytes that each thread or application occupies on the L3 cache. The same idea is applied to the bandwidth with the Intel MBM (Memory-Bandwith-Monitoring) technology, but in terms of bandwidth consumption. Finally, the Intel CAT (Cache-Allocation Technology) technology allows partitioning of the LLC.

Recent research [17],[14] that leverages Intel RDT distinguishes between three type of applications in relation to their memory and cache-access behavior:

1. **Streaming**: To this group belong applications whose working-set size is bigger than the LLC size and thus make intensive use of main memory. Not only that, but these applications do not have much locality nor a high reuse of information. Because of this access pattern, these applications will suffer from cache-misses, even with the mitigation pre-fetching can provide or many memory resources.

2. **Cache-Intensive**: Applications whose working-set size does not fit in the private cache levels. Another characteristic of these applications is that they are very sensitive to losing memory space, since they do not reuse much information brought from main memory either.

3. **Core-Fitting**: Applications that have a very low or no flow of information to upper levels of the memory-hierarchy as there is enough space withing the core's private cache levels. They show a high reuse access pattern.

It is no surprise that cache-intensive applications degrade their performance if streaming threads are executed simultaneously. In fact, threads can incur in more cache-misses if they share the LLC with "noisy neighbors" (see Figure 6.1). On the other hand, core-fitting applications are not affected nor affect the execution of the rest of threads in terms of memory.

In conclusion, when sharing the LLC by multiple applications the overall performance is unpredictable due to the possible contention that can arise per-application. Taking into account this in order to schedule the tasks can help improve the performance and avoid bottlenecks.

There is no consensus on what is the best way to take advantage of Intel RDT, there is no "right answer" so far. Several scheduling algorithms have being proposed [17],[19] that take advantage of the Intel technologies to improve scheduling, and their overall performance has been proved better than the default Linux scheduler (CFS) for many workloads.

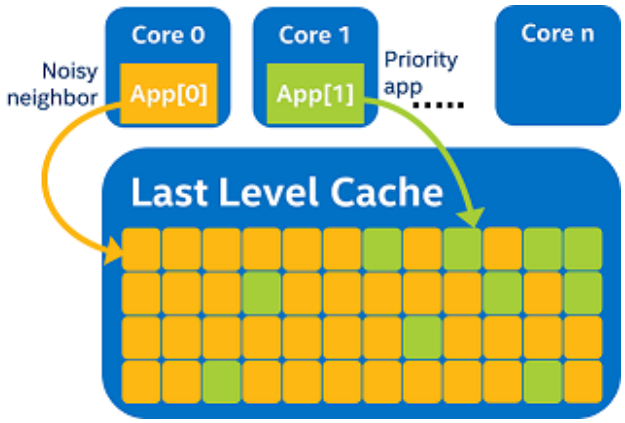


Figure 6.1: Noisy neighbor - Figure by Intel

As this project original goal was to provide support for the dynamic development of scheduling algorithms on the Linux kernel, it would be extremely useful to carry out some work to provide support for contention-aware scheduling.

Therefore, we believed it would be interesting to include support for such scheduling algorithms directly in the Linux kernel. The most straightforward way to achieve this is not by modifying the FSAC kernel, as starting from scratch for this purpose would be a way too ambitious goal for only one academic year. Instead, it was easier to add a new monitoring module into PMCTrack, which already had support for monitoring hardware counters and all Intel RDT, and -using the experience acquired with FSAC- develop a scheduling algorithm that could take note of the information regarding cache-occupancy, and that could be used as skeleton for contention-aware scheduling algorithms in the future.

This support for the Linux kernel development of cache-aware scheduling algorithms could be used in combination with the PBBCache Simulator [27], maintained by faculty at UCM, to

create new cache-partitioning algorithms and test them dynamically.

6.2 *n*-cores co-scheduling algorithm

One of the most important features that a framework that aims to provide support for memory-aware scheduling algorithms prototyping should have is co-scheduling support. PMCSched should include a proof of concept for such scheduling algorithms, for it could be of great help to developers. Not only that, but that this could be done in a modular way so that the programmer only needs to update a few details in order to **develop an algorithm with a completely different co-scheduling policy, by only changing the grouping criteria**, i.e. which threads are scheduled together. As a matter of fact, if we intend to develop a contention-aware algorithm we will need to use co-scheduling abstractions, so this scheduling algorithm needs to be done first in any case.

For all these reasons, we decided to build a basic co-scheduling algorithm. The proof of concept can be very simple, as we only need to fulfill that the plugin groups threads in sets of n-cores, being n the number of available cores. Our grouping criteria could be to give priority to those with smaller *vruntime*. This metric is included within the threads structure (*task_struct*) and provides an insight of the CPU usage of the given application. Not only that, but it also takes into account the priority of the thread. For example, two threads that run for x seconds on a core, will not have the same *vruntime* if they differ in priority. The higher the priority is, the slower the counter increases. Therefore, this proof of concept is a co-scheduling algorithm that takes into account both CPU utilization and priority.

In order to assign threads with smaller *vruntime* to cores, we decided to sort the list of stopped queues before the periodic scheduler kicks in. We do this with a new function for sorting the kernel linked lists that receives the list itself, a parameter to specify whether or not we want it in ascending order and a function to compare two threads. This is the basic function we came up with to compare the threads, but that can be easily changed by the developer to implement a completely different sorting policy:

```
inline int compare_threads(void* t1, void* t2) {
    u64 vruntime_t1, vruntime_t2;
    vruntime_t1 = ((sched_prot_thread_data_t*)t1)->prof->this_tsk->se.vruntime;
    vruntime_t2 = ((sched_prot_thread_data_t*)t2)->prof->this_tsk->se.vruntime;
    return vruntime_t1 - vruntime_t2;
}
```

As we want the list of stopped threads to be sorted in ascending order, we needed to return a negative value if `t1` is smaller than `t2`, zero if `t1` equals `t2` and positive otherwise. We compare the *vruntime* located at structure `sched_entity` from the *task_struct* of the thread. Then, we call in the periodic scheduler of the plugin the sorting function with the criteria as parameter:

```
sort_sized_list(&sched_prot_gbl.stopped_threads,1, compare_threads);
```

We had to face some other challenges involved in the development of a co-scheduling algorithm

inside the kernel. For example, we had to consider the scenario in which a thread leaves one of the idle cores earlier than expected, and so the new awaken task has to be placed at that same core. Hence, we had to keep track of the busy cores and the ones that went idle at runtime. In order to deal with this, we first created a function that traverses all the cores and flags the idle ones.

There is no need to traverse the list if there are no active threads. If there are, we start by marking all the cores as idle.

```
inline int mark_busy_cores(int *idle_cores, int num_cores){
    unsigned int i = 0, found = 0, cpu;
    sched_prot_thread_data_t *aux;
    if (!sized_list_length(&sched_prot_gbl.active_threads)) return 0;
    for (; i < num_cores; ++i) idle_cores[i] = 1;
```

After that, we iterate over the running active threads, and retrieve the CPU they are running at using kernel abstractions.

```
    aux = head_sized_list(&sched_prot_gbl.active_threads);
    for (i=0; i < sized_list_length(&sched_prot_gbl.active_threads)
        && (aux != NULL) ; ++i){
        if (aux->runnable && !signal_pending_th(aux) &&
            (aux->prof->this_tsk->state == TASK_RUNNING ||
            aux->prof->this_tsk->state == TASK_WAKING)) {
                cpu = task_cpu(aux->prof->this_tsk);
```

Note that the core the thread is located at may be already flagged, but in that case the number of idle cores should not be increased, as the running tasks can be in that processor run-queue. We just increase the counter if it is the first thread found running in a core.

```
            if (idle_cores[cpu]){
                found++;
                idle_cores[cpu] = 0 ;
            }
```

And finally at the plugin's periodic scheduling function we can compute the value of free spots and initialize the array in a single line:

```
free_spots = num_cores - mark_busy_cores(idle_cores,num_cores);
```

Once we know the number of idle cores, we can start assigning stopped threads to them. Nevertheless, it can also happen that a thread is already located at that CPU's run-queue and hence binding it would be a waste of time. Therefore, we prepared this other function to keep track of this at all the iterations to the periodic scheduler.

```
int next_free_cpu(int *idle_cores, int max){
    static int prev = -1, prev_2 = -1;
    int i = (prev > -1)? prev + 1: 0;
```

We could also need to undo one (just in case the thread was already there) or reinitialize for

next periodic call.

```
    if (!idle_cores) {
        if (max == -1) prev = prev-1;
        else prev = -1;
        i = -1;
    }
    else {
        for (; i < max && !idle_cores[i]; ++i){}
        if (i == max) i = -1;
        prev_2 = prev;
        prev = i;
    }
    return i;
}
```

Then, the periodic scheduler needs to initialize the counters, as well as going one step back if the thread being manipulated was already assigned to that core's queue, as previously explained.

```
if ((cpu = next_free_cpu(idle_cores,num_cores)) == -1) {
    (...) /* Error messages */ continue;
}
```

If the thread is already running on the desired core, we need to free that position of the array again. If not, the thread's affinity mask is cleared and set to the next free CPU available.

```
if (task_cpu(stopped->prof->this_tsk) != cpu){
    cpumask_clear(stopped->mask);
    cpumask_set_cpu(cpu, stopped->mask);
    insert_sized_list_tail(migration_list,stopped);
}
else cpu = next_free_cpu(NULL,-1);
```

6.3 Memory-cognizant algorithm *MEMSCHED*

Once the co-scheduling algorithm was developed, we had all the necessary code base to start working on the Memory-cognizant algorithm. This algorithm was proposed in previous work [17] but it was never tested at the kernel level. It intends to take into account the threads' memory behavior, in order to speed execution and improve fairness by reducing the memory contention, not only with less cache misses but also with the bandwidth occupation.

Algorithm 1 illustrates the behavior of the *MEMSCHED* scheduler. The *FITNESS* of a thread (how suitable would it be for a given scenario) in the algorithm is the metric described in the following formula:

$$FITNESS(t) = 1 \Big/ \left| \frac{Limit}{CPUremain} - Usage(t) \right| \tag{6.1}$$

where the values for the *Limit* and the *Usage* will be defined at runtime depending on the scenario, prioritizing bandwidth or memory occupancy.

Algorithm 1: Memory-cognizant scheduling algorithm

Input : n threads

Output: m threads to be executed on m cores (m <= n)

```
1  // PROFILING OF THE APPLICATION
2  Prepare a light-weight scenario, all threads are stopped but t
3  for t in threads do
4  |  Update IPC, BWmm, BWL3L2, USAGEllc in t
5  |  if BWmm > BWthreshold then
6  |  |  type(t) ←Streaming
7  |  end
8  |  if BWL3L2 > BWthreshold then type(t) ←Cache Intensive
9  |  else type(t) ←Core Fitting
10 |  Progress(t) ←Progress(t) + IPCcorunning / IPCalone
11 end
12 // ASSIGN THREADS TO CORES
13 Sort run-queue in ascending order with Progress
14 CacheOn ←StreamOn ←False
15 BWremain ←BWmm, LLCremain ←SIZEllc, CPUremain ←CPUs
16 for t in threads do
17 |  if type(t) = Streaming and not CacheOn then
18 |  |  StreamOn ←True
19 |  |  if BWmm < BWremain then BWremain ←BWremain - BWmm, CPUremain
   |  |   ←CPUremain - 1
20 |  end
21 |  if type(t) = Cache Intensive and not StreamOn then
22 |  |  CacheOn ←True
23 |  |  if USAGEllc < LLCremain then LLCremain -= USAGEllc, CPUremain
   |  |   ←CPUremain - 1
24 |  end
25 |  if type(t) = Core Fitting then CPUremain ←CPUremain - 1
26 end
27 if StreamOn then (Limit,Usaget) ←(BWremain, BWmm)
28 else (Limit,Usaget) ←(LLCremain, USAGEllc)
29 Sort run-queue in descending order by FITNESS(t) and update Limit ←Limit - Usaget
30 while CPUremain > 0 do
31 |  Select next thread t, CPUremain ←CPUremain - 1
32 end
```

In Algorithm 1, line 4 uses *IPC* for the Instructions Per Cycle measured for that thread, while *BWmm*, *BWL3L2* and *USAGEllc* respectively refer to the bandwidth consumption for main memory, the consumption between L3 and L2 and the usage of the Last Level Cache. Two flags are used to mark if there is already a streaming or cache-Intensive flag running (initialized on line 14). It is also important to note that on line 15 the values retrieved from the system specification are used to initialize the count of bandwidth, memory, and cores remaining.

Implementing this contention-aware algorithm (also referred to as *MEMSCHED* plugin in this report) was one of the most challenging parts of this project. First, we had to prepare the light-weight scenario in order to collect a profiling sample from the thread, running alone, so it could be labeled as one of the three categories previously described on Section 6.1. Nevertheless, we could not do that as soon as the thread enters the system because interruptions had to be enabled. For this reason, we added an extra linked list `pending_to_be_profiled` that any scheduling algorithm developed with PMCSched can use to insert the threads whose profiling are they interested in. Function `check_pending_profiling()` is periodically invoked by the PMCSched, and if necessary, function `light_weight_scenario()` is called.

During this stage of profiling, the periodic scheduling function is called more often, as we want to have the rest of the threads stopped as least time as possible. As soon as a performance sample is collected, normal execution is resumed. When that metrics are collected, the pseudo-code can be followed to label the thread. For instance, if the thread is of type STREAMING_TSK:

```
BWmm = t->BWmm;
if (BWmm > BWmm_thr){
    t->memory_profile = STREAMING_TSK;
    printk(KERN_INFO "%s: Thread %d labeled as STREAMING.\n",__func__,pid);
}
```

And now that the threads are profiled we can start assigning each of them to the cores. First, we need to sort the threads in ascending order by progress. We use the same notions discussed for the *n*-cores co-scheduling plugin but using a different grouping criteria (field `progress_p` at the *task_struct*).

After that, we can start iterating following the algorithm specification.

```
switch(activated->memory_profile){
    case STREAMING_TSK:
        if (!cache_flag){
            stream_flag = 1;
            if (BWp_MM < BW_remain){
                BW_remain -= BWp_MM;
                CPU_remain--;
                if (active_scheduler_verbose)
                    printk(KERN_INFO
                    "Streaming thread %d will be active.\n",
                    activated->prof->this_tsk->pid);
```

```
            }
        }
    case CACHE_INTENSIVE_TSK:
        if (!stream_flag){
            (...)
    case CORE_FITTING_TSK:
            (...)
```

Subsequently, we need to sort the list again, this time to try to maximize the FITNESS of the next selected thread, using the previously described equation. We need to find the processes that maximizes FITNESS(p), hence we need the list to be descending. We use the global variables `limit` and `CPU_remain`.

```
sort_sized_list(&sched_prot_gbl.all_threads,1,compare_fitness);

inline int compare_fitness(void* t_1, void* t_2){
    u64 usage_t1, usage_t2, fitness_t1,fitness_t2;
    (...)
    usage_t1 = (stream_flag)?t1->BWmm:t1->USAGE_LLC;
    usage_t2 = (stream_flag)?t2->BWmm:t2->USAGE_LLC;
    fitness_t1 = 1 / ((limit/CPU_remain) - usage_t1);
    fitness_t2 = 1 / ((limit/CPU_remain) - usage_t2);

    return fitness_t2 - fitness_t1;
}
```

Finally, we just need to assign each of the select threads to a core, using the same abstraction as in the n-cores co-scheduling algorithm and stopping the ones not selected to run for this iteration. In the next iteration threads will be relabeled in a different category with `update_threads_progress_relabel()` if some of them changed its memory-related behavior.

Let us review an excerpt of the debugging messages that are printed during the execution of some applications using this scheduling algorithm, with the verbose option activated. Two single-thread applications are launched, with PID 22325 and 22326. When they enter the system, a RMID (Resource Monitoring ID) gets assigned to them in order to use Intel RDT monitoring resources.

We can easily distinguish between different execution phases below. First, the thread is activated and the RMID is assigned (at ln.0). It is inserted in the list of threads that should be profiled and some time later removed from it when the light-weight scenario is prepared. In this case, nothing has to be done because it is the only thread running on the system. The performance sample is collected and the thread is labeled accordingly (ln.5). After that, a newly created thread with PID 22326 is activated and inserted into the pending list. This time, the light-weight scenario requires thread 22325 to be stopped and so a signal is sent. Finally, both threads are labeled and they can be scheduled. In this case, the assignment is straightforward as both of them are core-fitting and can use the CPU (ln.14).

```
[ 0] Thread 22325 assigned RMID 21.
```

```
[ 1] Newly active task 22325 will be profiled and stopped -- MEMSCHED.
[ 2] Task 22325 will be removed from pending profiles.
[ 3] Sample from thread 22325 collected at Light-Weight scenario.
[ 4] Thread 22325 registered BWmm = 128, BWL3_L2 = 0, USAGE_LLC = 128
[ 5] profile_thread_memory: Thread 22325 labeled as CORE FITTING.
[ 6] Thread 22326 assigned RMID 22.
[ 7] Newly active task 22326 will be profiled and stopped -- MEMSCHED.
[ 8] Task 22325 will be removed from pending profiles.
[ 9] __send_signal: wait_task_inactive thread 22325,state 0
[10] Task 22326 will be removed from pending signals.
[11] Sample from thread 22326 collected at Light-Weight scenario.
[12] Thread 22326 registered BWmm = 258, BWL3_L2 = 10, USAGE_LLC = 108
[13] profile_thread_memory: Thread 22326 labeled as CORE FITTING.
[14] Only a group of 2 threads scheduled. -- MEMSCHED
```

Chapter 7

Experimental Evaluation

This chapter covers the experimental analysis of PMCSched, one of the most critical aspects of this project. In it we explain the methodology, and discuss the collected performance data. The metrics obtained with the memory-aware scheduling algorithm are compared with the current Linux kernel scheduler.

Needless to say, the best way to test a framework is by making use of it. We developed several scheduling algorithms leveraging the API and abstractions provided by it. Some of them were just low-level implementations of well known algorithms, such as a variant of *Round Robin*. Nevertheless, one scheduling algorithm in particular was specially interesting to study, as it intended to introduce novel ideas that take shared resource contention into consideration. The performance results obtained from this scheduling plugin (referred to as *MEMSCHED* in advance) will be discussed now, along with metrics and charts, extracted from reports generated with dozens of benchmarks and by using custom scripts.

The experiments were conducted on a 64-bit x86 platform, featuring an **Intel Xeon CPU E5-2620 v4** that operates at a frequency of 2.10GHz. As shown in Figure 7.1, L3 cache memory had 20480 KBi L2 256 KBi and L1 32 KBi for instructions (L1i) and for data (L1d). During the experiments, Turbo Boost Technology and Hyper-threading were disabled.

To ensure covering a wide range of scenarios, **46 different benchmarks** from SPEC CPU 2006 and 2017 were employed, combining them in various proportions to generate diverse workloads. The questions that we intended to solve were as follows:

1. Does the PMCSched operate as expected? Its behavior should not interfere with the correct execution of the selected algorithm.
2. If it does, is the *MEMSCHED plugin* algorithm working as it should too? This is, the plugin should follow the provided pseudo-code specification [17].
3. If everything is working as it should, how good does the plugin perform? Is it similar to the current Linux scheduler? If not, why is it better or worst?

To address these questions, we carried out experiments with multiple *workloads*. A workload is a set of benchmarks running simultaneously on the system. In order to cover diverse

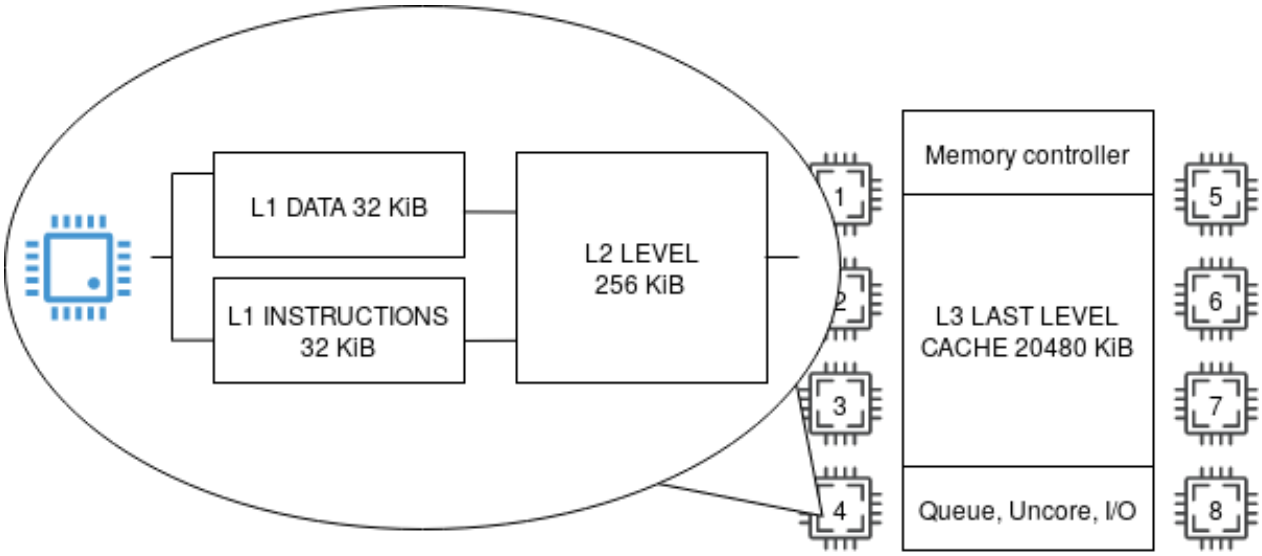


Figure 7.1: Memory Hierarchy of the platform used for the experiments

scenarios, **twenty-two workloads were built**. From these, the last six (Exp17-Exp22) sample extremely challenging conditions. In fact, experiments nineteen to twenty-two each have sixteen benchmarks, twice as many threads as the number of cores. The remaining workloads each include in most cases eight benchmarks.

It is important to remark that in those workloads composed of eight benchmarks, our evaluated scheduler (the *MEMSCHED* plugin) does not keep all cores busy all time (as there are eight cores). Instead, it takes into account their behavior towards memory resources, such as bandwidth and main memory. This may well result in situations where two or more tasks are not be allowed to run simultaneously, in order to reduce cache contention and other undesirable effects. In that sense, the algorithm is a non-work-conserving scheduler [28].

We tried to keep this in mind while preparing the experiments, mingling in different number of benchmarks that heavily use memory resources (such as *Streaming* applications) with those that do not (referred to as *Core-Fitting* applications). The way in which the plugin distinguishes between them at runtime is specified with a threshold that can be consulted and established via the `/proc/pmc/config` special file. In the experiments, this was set to 5% of the total bandwidth size (1677721KB from the 32GB available).

Figure 7.2 shows the composition of the various workloads. Colors indicate the level of memory intensity we expect from each application, and we use them to generate highly diverse scenarios. As it can be observed, Exp1 is fully *Core-Fitting* (green), while Exp2 contains *Cache Intensive* (orange) processes, and Exp3 is composed by only *Streaming* (red) applications. After these, mixes in different proportions are disposed. As explained before, to the *Streaming* group belong applications whose working-set size is bigger than the LLC size and have low locality, while *Cache Intensive* applications whose memory working-set size is approximately equal to the size of the LLC, and finally *Core Fitting* applications have a very low or no flow of information to upper levels of the memory-hierarchy.

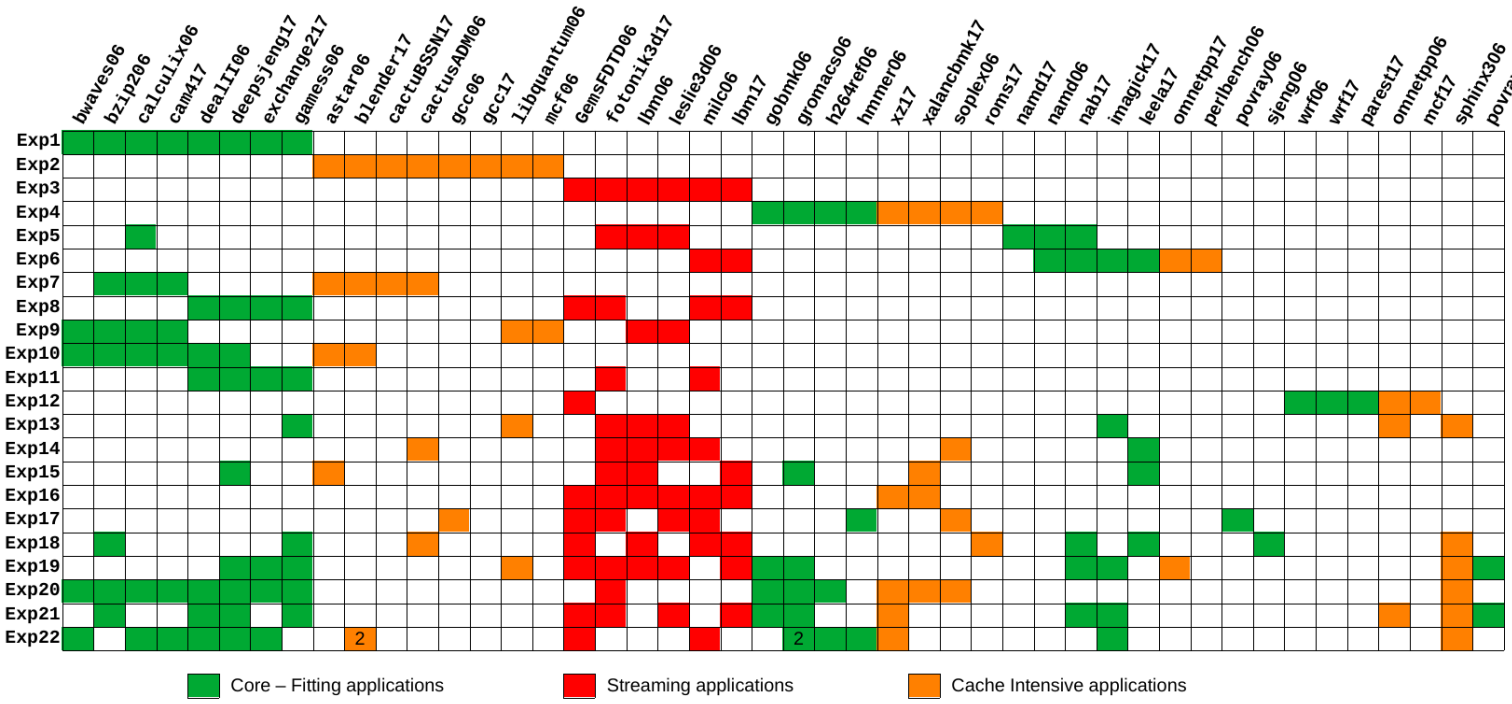


Figure 7.2: Experiments with their benchmarks

In addition, in Exp22 two instances of *Blender17* and *Gromacs06* are included in the workload. Once the experiments were conducted, several metrics were computed based on the turnaround time of each process. We will proceed to introduce them. All of the metrics we will discuss are based on the **Slowdown**, defined as:

$$Slowdown(app) = \frac{CT_{part}(app)}{CT_{alone}(app)} \tag{7.1}$$

where

$$CT_{part}(app) \tag{7.2}$$

is the Completion Time of the process as part of the workload and

$$CT_{alone}(app) \tag{7.3}$$

is the time it took it to finish alone. Most of the workloads tested had a negligible difference in terms of mean Slowdown of the applications (with thirteen of them less than a relative 4%).

The second metric we computed is the STP, System Throughput, defined as the sum of the normalized progress rate across all n benchmarks in the workload mix. In other words, the sum of the inverse slowdowns, as shown below:

$$STP = \sum_{i=1}^{n} \frac{1}{Slowdown(i)} = \sum_{i=1}^{n} \frac{CT_{alone}(app)}{CT_{part}(app)} \tag{7.4}$$

STP is the accumulated progress across all threads, and thus it is a higher-is-better metric that gives an overall idea of the system average throughput achieved.

The third performance metric we evaluate is the ANTT (Average Normalized Turnaround Time) the arithmetic mean across the programs' relative completion time. Hence, it is a lower-is-better value.

$$ANTT = \frac{1}{n}\sum_{i=1}^{n} Slowdown(i) = \frac{1}{n}\sum_{i=1}^{n} \frac{CT_{part}(app)}{CT_{alone}(app)} \tag{7.5}$$

To conclude, we are also interested in measuring how *unfair* the *MEMSCHED* algorithm is compared to the current Linux kernel scheduler. We can assume it will be less fair than it. That is because, while the Linux scheduler manages evenly threads with the same priority, our algorithm grants a preferential treatment to those with reduced memory usage, no matter their priority.

To measure the degree of fairness, we use the lower-is-better unfairness metric [6] defined as follows:

$$Unfairness = \frac{MAX(Slowdown(1), Slowdown(2)...Slowdown(n))}{MIN(Slowdown(1), Slowdown(2)...Slowdown(n))} \tag{7.6}$$

Regarding the System Throughput, we obtain very satisfactory results in the algorithm *MEMSCHED*, especially in highly memory intensive workloads. Figure 7.3 shows the normalized STP for the various worklads. **Exp19 experienced an STP improvement of 24% over the current Linux kernel scheduler**, and Exp22 of a 14%. Moreover, the worst result corresponds to Exp17 with a decrease of only 6% in system throughput. This decrease can be explained by reviewing the mean Slowdown of its applications, an eight percent slower on average compared to the Linux kernel scheduler.

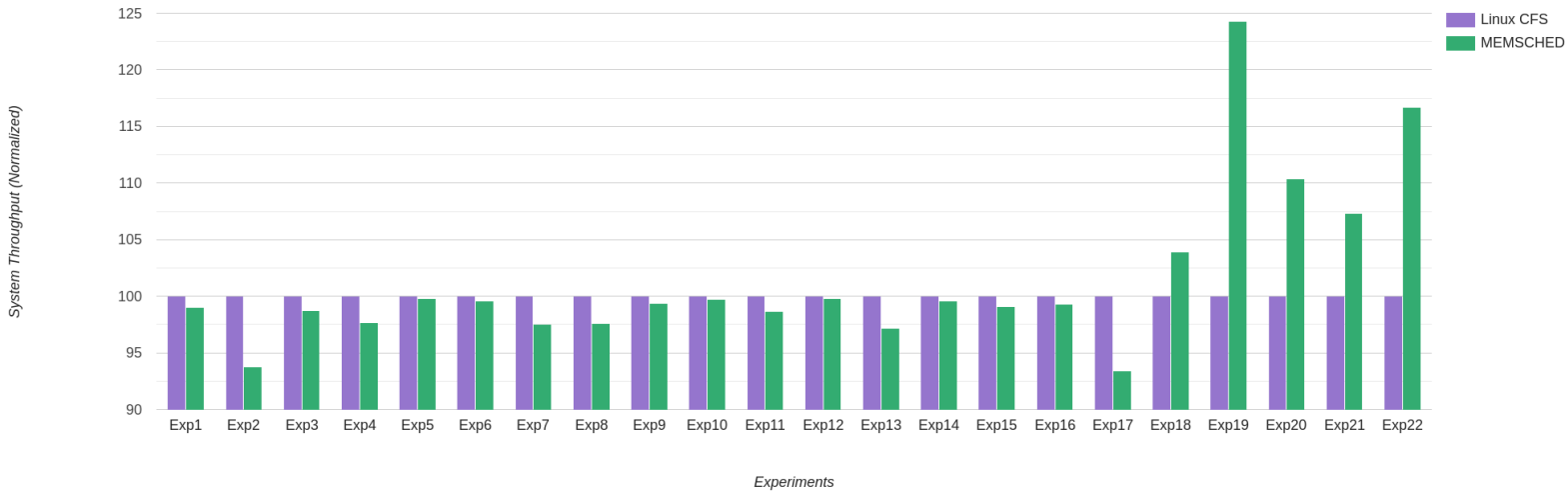


Figure 7.3: System Throughput (STP) compared to the Linux scheduler

For the normalized ANTT, values are displayed in Figure 7.4. Yet again, **Exp19 shows excellent results with an improvement of 15%**. The least desirable result (Exp17) is only a 10% worst than the current Linux kernel scheduler. The least preferable values obtained from Exp17 can be understood by the nature of the benchmarks employed on it, a very high proportion of them being *Streaming* applications.

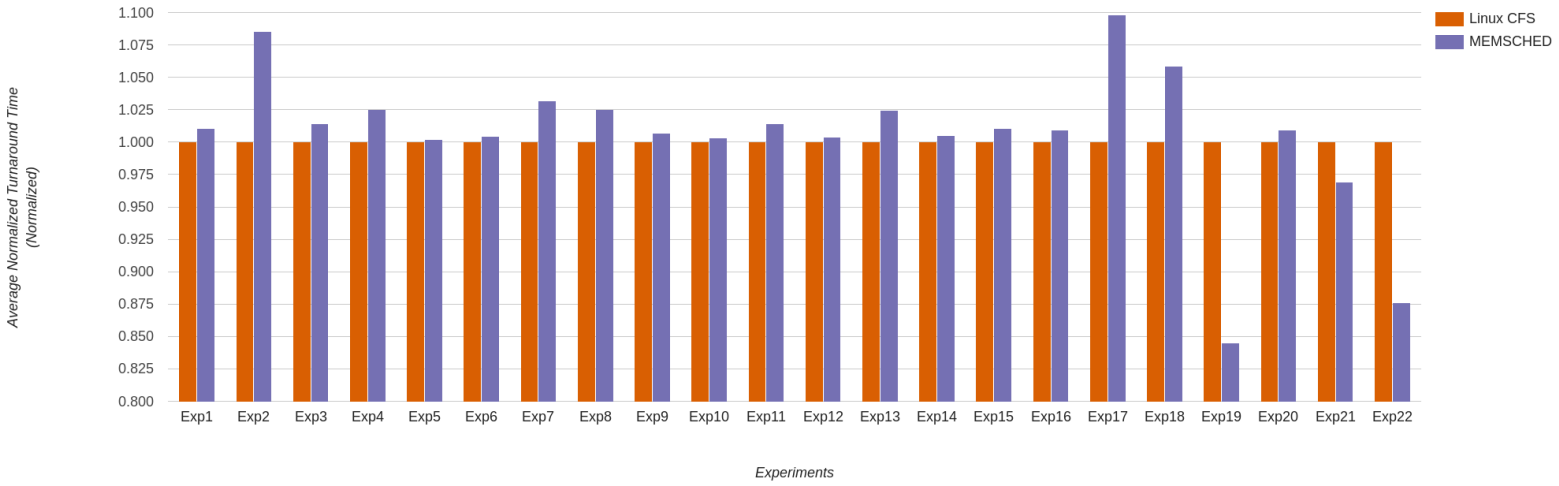


Figure 7.4: Average Normalized Turnaround Time (ANTT) comparison

Figure 7.5 below shows the comparison in unfairness between the current scheduler and the *MEMSCHED* algorithm. We divide our experiments into two sets to make it easier to see the details. For the first set, the computed unfairness is negligible in most cases, since all of them are less than %10 more unfair, exceptions being Exp3 (with 27%) and Exp7 (with 13%). This is no surprise, since as we explained before, Exp3 is composed by fully *Streaming* applications (Figure 7.1), hence the algorithm *MEMSCHED* will not allow many of them to run simultaneously.

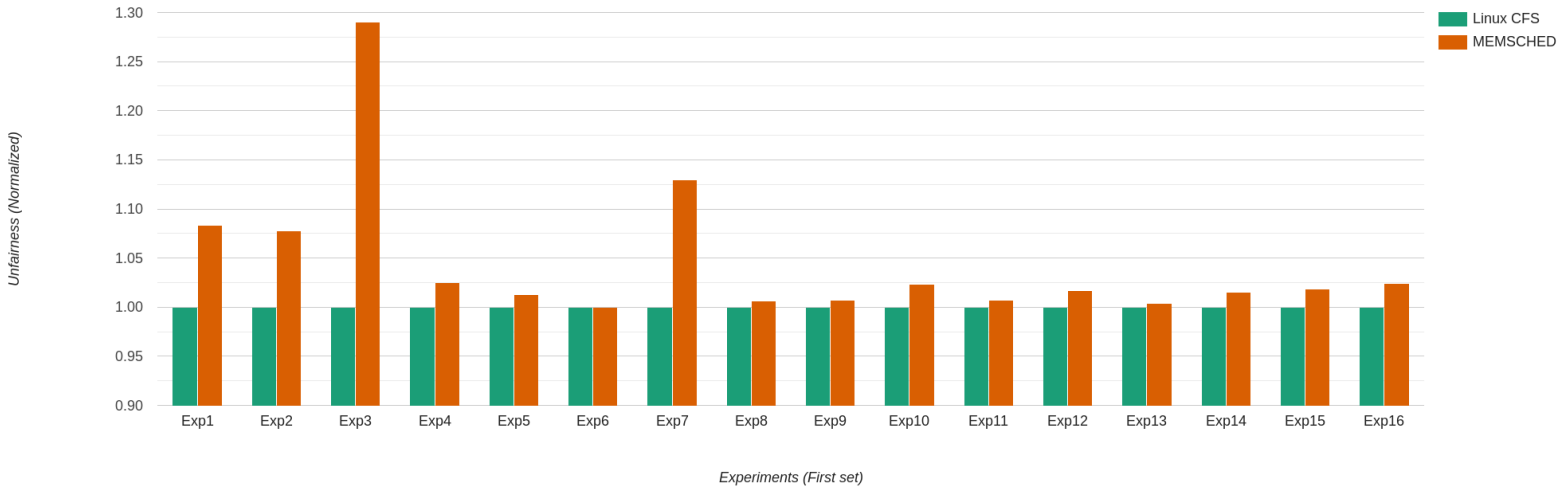


Figure 7.5: Unfairness of MEMSCHED compared to the Linux kernel (first set)

The second set of experiments suffers from greater unfairness. As we include more benchmarks, the algorithm will give preference to the ones with lower degree of memory intensity. All the experiments in this set experience under a 30% increase in unfairness, exceptions being Exp20 (75%) and Exp18 (70%). Although worst fairness results are obtained, the overall throughput improves significantly, as shown before.

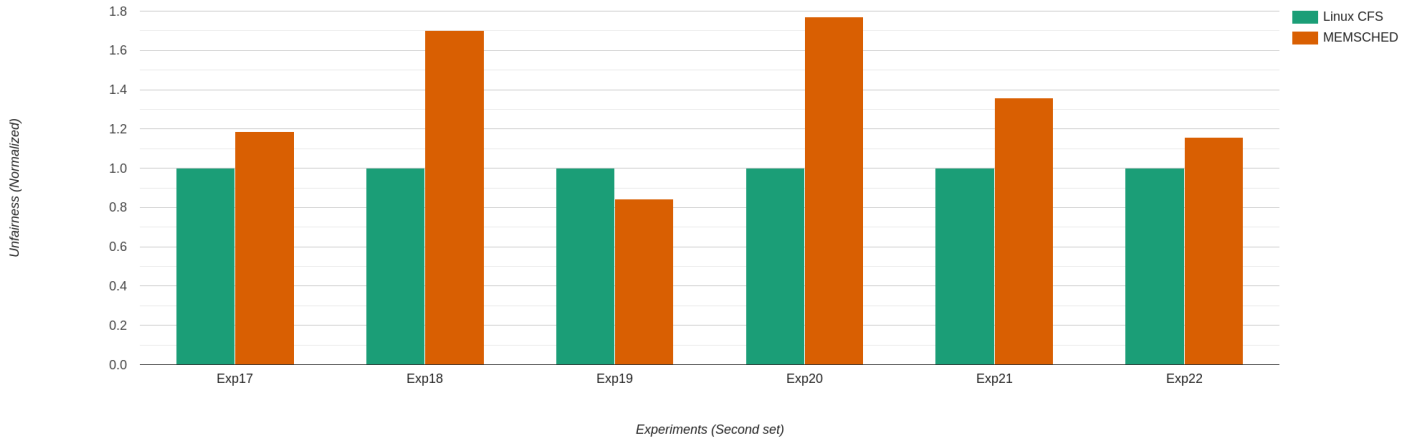


Figure 7.6: Unfairness of the algorithm compared to the Linux kernel (second set)

7.1 Traces from reports

After analyzing the performance metrics, it is also of great interest to obtain traces from the scheduler's behavior during benchmarks' execution. In other words, a graphical way to observe which thread runs at all times. In the results shown in the previous section only the completion time of each application was gathered. However, they do not reflect periods of inactivity, or inform about what benchmarks were scheduled to run simultaneously and when. Turnaround metrics alone fail to provide with a comprehensive view of the scheduler's inner workings.

In order to build charts that displayed the state of the threads during the workload execution, it was necessary to include trace points in the Linux kernel to register important events and write them into a log that could be later transformed into a chart.

For that purpose, a script in the SystemTap language was developed. SystemTap is a scripting language and a powerful OS tracing and inspection monitoring tool. Examples of things it can trace include context switches, thread migrations or module functions invocations. The script *process_tracer.stp* created during this project uses associative arrays to store each of the relevant events, such as invocations to *MEMSCHED*'s function `switch_in()`. Reports generated with that script follow the .prv file format [18], in order to leverage the Paraver software later on. The software is developed and maintained by the Barcelona Supercomputing Center.

For example, Figure 3.7 shows the generated graph for the Exp19 with gradient color for CPU consumption time. We provide information for each of the threads on Table 7.1.

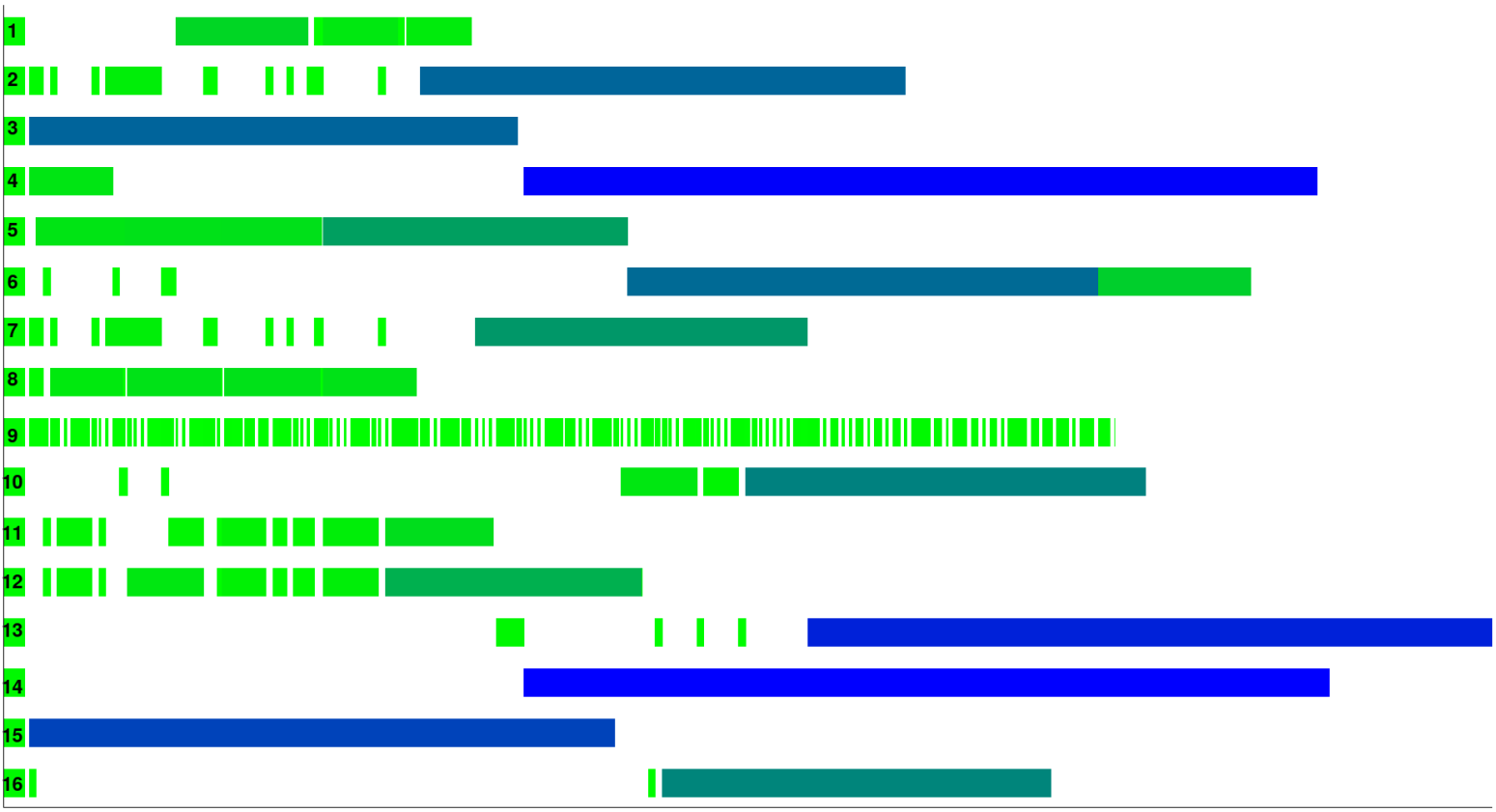


Figure 7.7: Useful duration of threads for Exp19

Table 7.1: Benchmarks of experiment 19

Thread number	Application name	Application category
1	Gamess06	Core-Fitting
2	Nab17	Core-Fitting
3	Exchange217	Core-Fitting
4	Imagick17	Core-Fitting
5	Fotonik3d17	Streaming
6	Leslie3d06	Streaming
7	Gromacs06	Core-Fitting
8	Gobmk06	Core-Fitting
9	Povray17	Core-Fitting
10	Libquantum06	Cache-Intensive
11	Lbm17	Streaming
12	Omnetpp06	Core-Fitting
13	GemsFDTD06	Streaming
14	Sphinx306	Cache-Intensive
15	Deepsjeng17	Core-Fitting
16	Lbm06	Streaming

These representations allow us to better understand the algorithm's behavior and make sure it is working as intended. For instance, at a first glance Figure 3.7 reveals that thread 13 is not allowed to run almost until the very end. This makes sense, as thread 13 was benchmark *GemsFDTD06*, cataloged as Streaming (view Figure 7.1) and thus it has to wait for others that make less use of memory resources. Same thing occurs with thread 11 Streaming application *Lbm17* and thread 6, that corresponds to the Streaming benchmark *Leslie3d06*. On the contrary, Core-Fitting processes are allowed to run from the very beginning. Such is the case of thread 15 *Deepsjeng17*.

The general picture seems to be satisfactory, but let us take a closer look into a specific moment of time, and see if what was going on in the CPUs checks out. A red line indicates the specific time instant we will proceed to discuss.

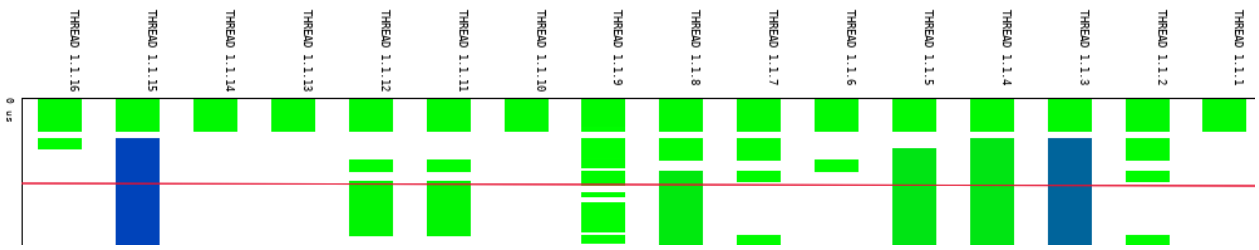


Figure 7.8: Red line over instant in time for Exp19

Let us analyze the threads running simultaneously in that specific moment, at an early stage of the experiment. It is important to note that the scheduling algorithm will force each thread to run on a different core using migrations via affinity masks. Looking at the previous table and the red line we can see that most of the threads (3,4,8,9,12,15) are Core-Fitting. The only exceptions are thread 5 (*Fotonik3d17*) and thread 11 (*Lbm17*).

We can clearly notice this algorithm favors processes that do not need to compete for memory resources. Also, there are no more than eight threads running simultaneously, which would have meant a tracing error. Even though there are two Streaming benchmarks are running, this will only happen briefly as the algorithm will preempt *Lbm17* soon after. Likewise, they running simultaneously momentarily is not an error, since the algorithm also evaluates the relative progress of the thread and there was no Cache Intensive application running at that time.

We conclude this chapter by answering the questions formulated at the beginning:

1. **Does PMCSched operate as expected?**

   Yes, it does. We have tested it both by developing a new algorithm and by running experiments that heavily stress the systems' computational and memory resources. Due to unpredictable scenarios, some bugs were detected during the experimental stage, but they were fixed effectively making the final framework more robust.

   In addition, we can conclude that the overhead introduced by the framework is negligible. There are virtually no statistically significant performance differences between running several threads one after the other or doing so by executing all with our special *FIFO*

plugin. The main overhead we faced was consequence of the debugging messages printed on the kernel buffer, which can be easily avoided by disabling the verbose option via terminal with `echo 'verbose 0' > /proc/pmc/config`.

2. If it does, **is the *MEMSCHED plugin* algorithm working as it should too?**

   Yes, the plugin follows the desired pseudo-code specification and does not show an unexpected behavior. As we have discussed with the Paraver diagram, core-fitting threads are given a preferential treatment, but other threads are not completely left behind in order to provide some level of fairness.

   Several improvements could be added in future versions of the algorithm. For example, we found the *profiling* stage to be unnecessary in certain situations. A new sample of the thread's performance will be collected very soon and its category will be updated, so there is really no need to perform such heavyweight profiling tasks in advance, that require to prepare light-weight scenarios. We will talk about profiling again in the Conclusions Chapter.

   Furthermore, **we find that Profiling deteriorates fairness in ways never discussed before**. This is, while the first thread to start running will be stopped for each and every other profiling, the last thread will never have to wait for such a thing. This difference can be substantial at times, especially for large workloads.

3. If everything is working as it should, how good is the plugin performance? Is it similar to the current Linux scheduler? If not, why is it better or worst?

   In small workloads the difference is not significant. In more aggressive scenarios, although a worsening in the mean Slowdown can be experienced at times, **the measured STP improves significantly**.

   In conclusion, it is fair to state that this algorithm is neither better nor worse than the one in the Linux kernel, but that there are situations in which its use would be preferable. These types of situations would be those that include many workloads. It is important to emphasize that the more benchmarks we run, the greater the possibility that we will have two or more applications that use memory resources intensively running together.

Chapter 8

Conclusions and Future Work

New application domains such as Big Data or Machine Learning acceleration have put hardware in check. Surprisingly, for the first time in many years hardware is losing the race to the market demands. Trying to cope with new application domains is often too computationally demanding, or plainly impossible for the computer one could expect to find in a regular home. As a result, only the biggest tech companies can afford to collect, analyze and extract meaningful information from this data using their own resources.

At the same time, it is becoming harder and harder to improve hardware, at least at the speed the semiconductor industry was used to. In fact, only a few companies are still trying to fulfill the Moore's prophecy, even though they are facing increasingly complex challenges. Hitherto, some researchers have started referring to this period as the *Post-Moore Era* [1].

But we do not need to lose all hope. In the upper layer (the operating system) many things have not being improved much for decades, relying on the improvement of hardware or higher layers of the software stack. There are countless possibilities for improvement in that direction.

The main problem is that creating and optimizing software at the OS level is far from being easy. A deep understanding of both the hardware that is going to be managed and of the applications that will try to make use of it is required. For instance, one cannot write a driver for a drone without being familiar -at least- with its custom firmware in the microcontroller and at the same time with the interface the user will use to send commands to the robotic vehicle.

In addition, there is a need of improving the scheduling algorithms present on current systems, which do not fully take into account many factors. One of them is the performance degradation that arises due to shared resource contention: On multicore systems, applications may heavily compete for the use of shared resources [3], such as the DRAM controller, the last-level cache and the bus or interconnection network. In addition, this leads to unfair situations as some applications make more intensive use of these components.

Based on these observations, the main goal of this project was to ease the work of OS developers trying to improve a specific part of the Linux kernel, the scheduler. This is one of the most critical components since it is in charge of distributing CPU time among threads and

performing thread-to-core assignments. To accomplish the aforementioned goal, the FSAC (*Framework for Scheduling Algorithm Creation*) kernel project was started. At the same time, we developed PMCSched to add support for scheduling algorithm prototyping in PMCTrack [7].

In order to address the OS development challenge, we started the **FSAC** project, a variant of the Linux Kernel that helps dealing with the general design and prototyping of scheduling algorithms. This framework avoids hours of kernel recompilation and allows the dynamic patching of the scheduler without rebooting the system, as it can be done at runtime. Its development was based on a previous project LITMUS-RT [5], which did not allow dynamic prototyping but had some useful abstractions already implemented. All the prior work and our contributions to that project were covered in Chapter 3 "*Prior work and LITMUS-RT patches*".

Secondly, other framework referred to as PMCSched was developed inside the open-source monitoring tool PMCTrack, to allow the prototyping of resource-conscious scheduling algorithms. PMCTrack already had all the support required for monitoring memory-related events, leveraging Intel advanced monitoring and hardware allocation technologies [8]–[11], as well as access to other hardware monitoring information (such as energy consumption). Summing both with patches to related projects, we can conclude this project involved more than **six thousand lines of kernel code**.

PMCSched and the FSAC kernel are both OS-level frameworks for rapid prototyping of scheduling algorithms. That being said, it is extremely important to keep in mind that they had very different implementations and most importantly they do not share purpose nor advantages. While the FSAC kernel offers a very precise and fine-grained control for efficient implementations -at the expense of a higher development effort- PMCSched has been optimized for co-scheduling and algorithms that require the scheduler leveraging information from performance monitoring. With FSAC the algorithm has absolute control over the run-queues in the cores and higher flexibility, but PMCSched can take advantage of several utilities of PMCTrack, such as the selective scheduling of a subset of tasks or the system information collected from the monitoring modules at runtime.

Once the development of PMCSched was completed, we decided to test it and created a contention-aware scheduling algorithm with it. This way, we could test the robustness of the framework and experiment with novel ideas that were recently proposed to deal with the shared-resource contention problem.

The results obtained with the *MEMSCHED* scheduling algorithm developed with PMCSched were satisfactory. In particular, we measured a significant improvement in system throughput (STP) of up to a 24% in comparison with the current Linux scheduler. Almost **50 benchmarks were employed in our experiments** . With a script to generate execution traces that we developed, we were able to compose charts with the values that provided with interesting insights of the behavior of the algorithm.

8.1 Future Work

A single user can run many different workloads on the same computer. For example, that person could be eager to play online videogames, editing pictures or perhaps be interested in watching video on streaming. All these things are achieved with applications that the operating system has to manage underneath, and has to provide with as many resources as possible.

Each of the aforementioned examples have very little in common, and the processes involved in their accomplishment have very different requirements. Presumably, videogames need to use the networking hardware frequently, while the image processing is very GPU demanding. Some of them will use memory resources very intensively and others will have enough with the space provided in the processor's private cache levels.

Yet, nowadays all of them would receive a similar treatment by the operating system, unless some priority constraints are explicitly specified. The kernel is to this day transparent to all these necessities and this undoubtedly impacts performance. The Linux kernel is present in every Supercomputer listed in the TOP 500, yet it only contemplates three possible families of applications: Hard Real Time, Soft Real Time and normal.

There are many possibilities to improve the scheduler and make it conscious about the particular needs of applications. With a framework like PMCSched, IT professionals and scientists could even use systems with ad-hoc schedulers conceived for their specific requirements.

For instance, companies with a high demand of AI training could use a scheduler specifically designed to improve performance for that workload type. Maybe -and this is just a hypothetical example- researchers find that some type of AI threads are more suitable to run in parallel to threads of some other type, and that AI-training threads belonging to a third type should run alone, all resulting in less cache-misses, bandwidth contention and eventually finishing execution earlier.

Regarding possible **improvements for this project**, PMCSched and the developed *MEMSCHED* algorithm could indeed be improved. In fact, we believe that the *FSAC* kernel was made modular enough to keep improving it in the future without much effort. The most straightforward new features that could be incorporated into FSAC at this point would be a set of wrapper scripts to interact with the `/proc`-based management interface.

Regarding the PMCTrack extension, PMCSched could do even more for the developer. So far, the API for developing scheduling algorithms does not provide abstractions to partition cache with Intel CAT technologies [12]. In addition, the frequency in which the periodic scheduling function is invoked was set arbitrarily, while other yet unexplored values could yield better performance or fairness.

Other potential improvement would be to add extra support to automatically retrieve system information, such as the maximum supported bandwidth, instead of hard-coding these values or asking the developer to manually provide the associated values.

For the contention-aware algorithm *MEMSCHED* some improvements could be carried out as well. The threshold to distinguish applications and categorize its memory-related behavior

was set arbitrarily at some extent too. It was assigned after reviewing the results measured from several benchmarks with previously known behavior. Nevertheless, the algorithm could surely benefit from a more precise value of the threshold. In addition, the complexity of the algorithm could be improved from $\mathcal{O}(n \log n)$ to $\mathcal{O}(kn)$ by modifying the way we handle doubly linked lists. Finally, it would also be interesting to consider that profiling threads -as explained in the Chapter 6- increases *unfairness*. This is because, while the first thread to be managed will have to be stopped for any new incoming one, the last thread will not have to wait for any other's profiling stage.

A. Installation and replication guide

We believed it would be interesting to discuss the necessary work environment and methodology, as a way of start digging into this project and making replication possible. We strongly encourage the reader to review this subsection if he/she is interested in replicating some of the work carried out here, installing FSAC Kernel, or is just looking for some extra hints on how to manage software development when it comes to different OS Kernel versions. Should that not be the case, feel free to skip this subsection.

The work environment will be now briefly described, and its usage should be at least considered by students desiring to start in the vast world of the Linux Kernel development or other low-level operating systems issues as it is the FSAC Kernel.

Compiling a Kernel is a highly computationally-demanding task. An efficient system is of great help for saving time and resources, but current laptops rarely fulfill such fine specifications. Thereby, we connected via SSH to a remote Debian with better characteristics.

Once it is compiled and the image is installed, rebooting the machine and choosing the recently added Kernel from the Grub list is also necessary. Needless to say, an SSH connection is not suitable for that purpose. Consequently, we used a second Debian machine with a SoL (Serial over LAN) connection to an Ipmitool (Intelligent Platform Management Interface) configured into the first one. Thus, it allowed me to remotely boot the first system leveraging a Kernel device driver. An alternative would be to change the default OS by editing the Grub configuration file (like with *grub_customizer*), but doing so we would have missed the boot messages, and we would not have been able to check the buffer afterwards if our Kernel had crashed before initialization.

If after trying to boot the kernel you encounter an error, a possible approach to debug is using `gdb`. If what is failing is a kernel module, its file `.ko` can be passed as an argument to `gdb`. After disassembling, you can find for the offended line adding the offset given by the *kernel panic* to the calling address of the function.

As a side note, we would like to point out that IPMI has been found to be extremely vulnerable and therefore you should configure it using only your private network. For an additional precaution we recommend, if possible, disabling IPMI from the BIOS as soon as you are done using it. Other auxiliary tool that was of great assistance is GNU Screen, as you can resume detached screen-sessions with paused processes. An alternative to IPMI is *minicom*.

This is an example of a remote connection from the second machine (via SSH) into the serial

port:

```
$ ipmitool -v -U Carlos -I lanplus -H machinesc sol activate
Password:
Discovered IPMB address 0x0
[SOL Session operational.  Use ~? for help]
(...)
```

Alternatively, if you are going to compile your Kernel versions using some external provider or system, instead of using plain SSH you can use sshfs as shown below. Sshfs is a file-system client based on SSH.

```
$ sshfs user@remote_machine:. ./new_folder_ssh
$ ls
$ xdg-open new_folder_ssh
```

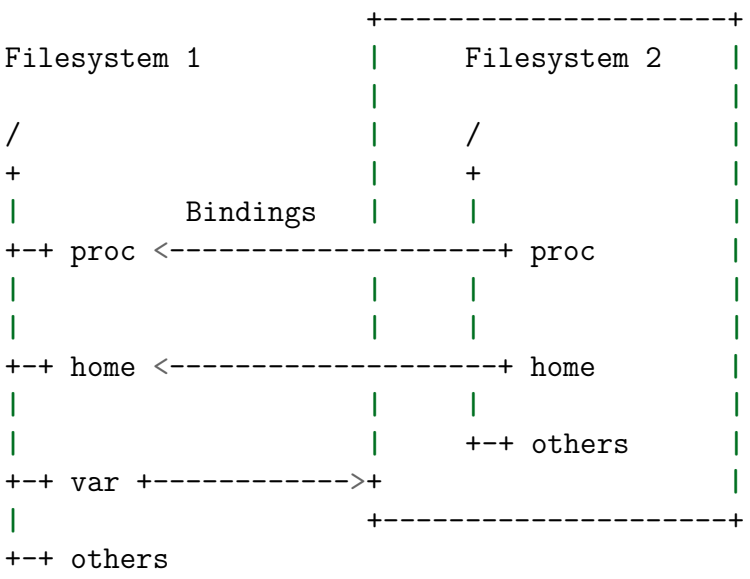
The constant upgrading of libraries and other resources by package maintainers is other important issue. Indeed, one of the GNU/Linux strengths can actually become a drawback for Kernel developers. For instance, when working in a fairly old project (LITMUS-RT), the GCC compiler was accidentally upgraded into a version that added tons of unnecessary -at least for our necessities- new warnings and errors that prevented compilation. Of course, there are easy-to-fix cases. For example, we could solve one error by updating the Makefile. However, other errors can be way more tricky.

Therefore, it was useful for me to use **Deboostrap**. It is a tool that mounts a Debian base system into a subdirectory of another -an already installed system- which does not have to necessarily be Debian. In this way, we had a non-changing version of the system inside the other one, jailed, via file system's binding.

With a script that mounts the binded file system, we could finally move between versions comfortably.

```
$ cat /etc/debian_version
10.0
 # Mounts and bind (mount -o bind /proc,/sys,/dev/tmp)
$ debian-login.sh
[sudo] password for Carlos:
$ cat /etc/debian_version
9.11
# You could also use Ctr+D
$ logout
$ cat /etc/debian_version
10.0
```

Notably, thanks to the previous commands, we can -for instance- assume that folder `/etc` is also binded, even-though it does not appear in the diagram next page, which represents the final structure.



After dealing with that issue, everything is prepared to start with the Kernel development and compilation. Hence, we will show now the complete methodology to install our new Kernel from the directory of its source code.

Firstly, we need a configuration file to install our Kernel. The fastest way is to copy a previously existing one. We included one in FSAC's repository.

```
$ cp /boot/default-config   .  # Copy some previous configuration file
```

We suggest reducing the size of this configuration file according to the project requirements. For instance, at some points we did not need anything related to GPUs or special debugging information, which introduce much overhead and augment compilation time. Hence, setting *CONFIG_DEBUG_INFO=n* is probably a good choice, unless some special low-level diagnosis is desired as in SystemTap scripts. If you want to use SystemTap, enable that option and then install the *image_dbg* depuration symbols generated. You can also use *kprobes* for some of the most basic symbols as well as other alternatives that leverage BPF Berkeley Packet Filter [29].

```
$ make oldconfig    # Or make menuconfig for a visual inspection
$ make -j15 bindeb-pkg LOCALVERSION=-example_patch # We omit the source package
```

The previously shown command is an example of the compilation colloquially referred to as as 'Debian style', which is faster than the traditional approach. Also, if the system supports hyper-threading, the *-j* option can be of great help to speed-up compilation. You can check if your cores mark the 'ht' flag with `$cat /proc/cpuinfo`. Be aware though that hyper-threading might complicate scheduling algorithms development.

If everything went as expected, you will encounter in the upper directory something similar to this. FSAC already contains files as these, packed in the traditional Debian style (.deb).

```
linux-firmware-image-4.9.30-example-patch_4.9.30-parche-patch-1_amd64.deb
linux-headers-4.9.30-example-patch_4.9.30-parche-patch-1_amd64.deb
```

```
linux-image-4.9.33-example-patch-4-9-33_4.9.33-parche-pmc-4-9-33-2_amd64.deb
linux-libc-dev_4.9.33-example-patch-4-9-33-2_amd64.deb
```

After this, the only thing left is the installation of the Linux image. Then, we can reboot and choose our new FSAC Kernel from the Grub List, using Ipmitool if we are connecting to a remote machine.

```
$ logout            # Back to the main Debian (with a root user)
$ sudo dpkg -i ./Carlos/Kernel/linux-image-*.deb
$ shutdown -r now   # Restart
```

Once rebooted, two methods are typically used to compile Kernel modules before insertion. The first one is to directly compile the new Kernel module "*against*" our custom Kernel.

```
$ make KERNEL_TREE=/home/Carlos/our_Kernel_folder
# Having at Makefile: "make -C $(KERNEL_TREE)  M=$(PWD) modules"
```

An alternative to this is installing the headers. In that case, we would need to use the main OS -where we can be root if binded-, but as we want to do so for the jailed Debian, we need our actions as superusers to affect the inner system. Therefore, we need to change our root directory before, a quite handy trick. We personally recommend this option as this will include a directory in `/lib/modules/` with symbolic links to `/usr/src/linux` for the modules, that some scripts within the jailed OS might attempt to use. Please note that installing these headers does not suppose to update `/usr/include/linux`, headers used for compiling libc. **You can upgrade your kernel to whatever version, up and backward, without touching user-space**.

```
# ls
bin  boot  dev  etc  home  lib  (...)
# dpkg -i ./home/Carlos/Kernels/linux-headers-4.9.30-patch-*.deb
(Reading database ... 68559 files and directories currently installed.)
Preparing to unpack
 .../linux-headers-4.9.30-parche-patch_4.9.30-patch-1_amd64.deb ...
Unpacking linux-headers-4.9.30-parche-patch (4.9.30-patch-1)
over (4.9.30-parche-patch-1) ...
Setting up linux-headers-4.9.30-parche-patch (4.9.30-patch-1) ...
#
```

CAPÍTULOS TRADUCIDOS AL ESPAÑOL

Chapter 9

Introducción

La justificación de todo el esfuerzo realizado en este Trabajo de Fin de Grado requiere describir ciertas tendencias actuales del mercado y dos de los retos tecnológicos más importantes que se deberán afrontar: El prototipado de algoritmos de planificación y el problema de contención de memoria. Juntos demuestran que el kernel de Linux adquirirá un rol decisivo en los próximos años de la Ingeniería Informática. En este Capítulo apoyaremos esta afirmación y explicaremos el proyecto, que está estrechamente relacionado con las mencionadas tendencias y diversos esfuerzos científicos actuales. Así mismo, detallaremos el plan de trabajo. Por último, expondremos en un resumen completo los contenidos de la memoria.

El proyecto entero se divide en dos partes y se ha estructurado esta memoria acorde a ello. En la siguiente sección se identificarán dos importantes retos tecnológicos y explicaremos cómo se ha decidido afrontarlos. Ambos están directamente ligados a la investigación en Sistemas. Dicho esto, es razonable afirmar que muestran potenciales implicaciones en otros campos.

9.1 Motivación

9.1.1 Los problemas Memory Wall y Power Wall

Uno de los mayores retos en Arquitectura de Computadores es el conocido como **Power-Wall**. Los límites físicos de la tecnología CMOS están a punto de ser alcanzados, y ahora más que nunca es importante tener en cuenta la eficiencia energética. De hecho, las dificultades técnicas asociadas a la producción a escala actual aumentan los costes de fabricación. A día de hoy, solo tres compañías intentan aún cumplir la profecía de Moore (Intel, Samsung Electronics y Taiwan Semiconductor Manufacturing Company Limited o TSMC), aunque estén afrontando un innegable declive en rentabilidad económica y un descenso en capacidad de integración, como demuestra la Figura 1.1. En consecuencia, algunos investigadores han comenzado a referirse a este periodo como la **"Era Post-Moore"** [1].

Nuevas tendencias han surgido en el mercado debido a estas conocidas limitaciones. Actualmente, algunas empresas intentan integrar el máximo número de núcleos por chip -manteniendo

frecuencias moderadas del procesador- mientras otras combinan cores y/o aceleradores con diferentes especificaciones dando resultado a una amplia variedad de **arquitecturas heterogéneas** [2].

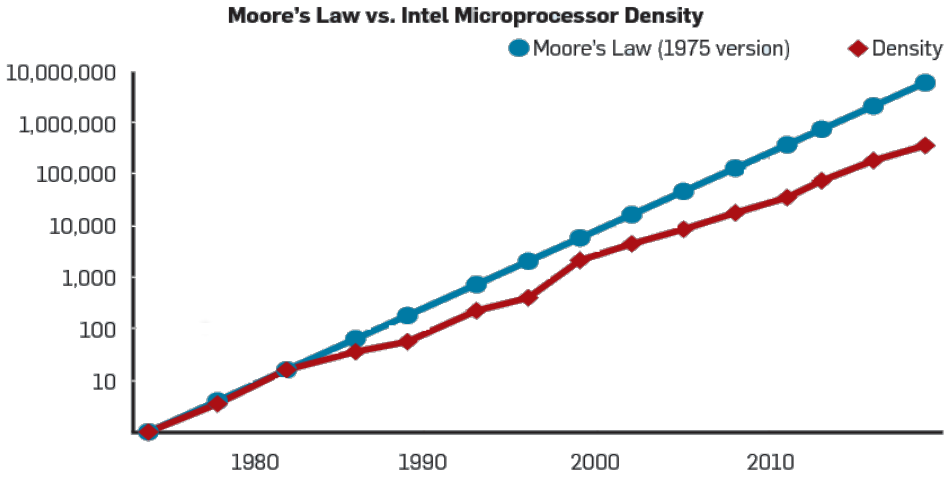


Figure 9.1: Predicción realizada en 1975 por Gordon Moore, en contraste con el número de transistores por chip de los procesadores Intel [1]

Sin embargo, ambas estrategias plantean retos importantes. Con la primera vía de desarrollo (arquitecturas multinúcleo simétricas), el creciente número de cores por CPU agrava el problema de la contención de recursos. En particular, todos los procesos ejecutados simultáneamente son forzados a competir por recursos de memoria [3], tales como el controlador DRAM, el último nivel de caché y el bus o red de interconexión. Como resultado, la contención de recursos de memoria agrava el problema **Memory Wall**. Las arquitecturas heterogéneas parecen más prometedoras en cuanto a eficiencia energética. No obstante, presentan también importantes limitaciones en términos de flexibilidad de la programación, ya que deben administrar diferentes Instruction Set Architectures (ISAs). Una alternativa para evitar esto son los **Asymmetric Multicore Processors** (AMPs), que combinan cores grandes y complejos con otros más eficientes energéticamente que tienen menor coste energético asociado. Son muy populares en telefonía móvil. Por citar un ejemplo, la familia de procesadores big.LITTLE de ARM [4].

En conclusión, no cabe duda de que el mercado muestra una tendencia a la especialización. La heterogeneidad ha demostrado ser la solución más viable y hay una creciente demanda que requiere de respuesta por parte del mundo académico que investiga en Sistemas. Los principales esfuerzos de este proyecto se centraron en adaptar GNU/Linux y en particular su kernel a esta tendencia emergente. Dicho en pocas palabras, la intención era facilitar el trabajo de los desarrolladores de sistemas operativos que deberán solventar problemas relativos a la memoria en arquitecturas multicore.

9.1.2 El reto de desarrollar en el SO

El Sistema Operativo GNU/Linux es utilizado por millones en una variedad de distribuciones. Se puede encontrar en todas las formas y tamaños, desde versiones centradas en seguridad (Kali Linux, Parrot, BlackArch,…) hasta otras extremadamente ligeras que caben casi en cualquier plataforma (Linux Lite, Ubuntu Mate, …). Google ha invertido mucho en sus Chromebook y Android -que se basa en el kernel de Linux- continúa siendo líder absoluto en telefonía móvil a nivel mundial.

Lo principal que todas estas distribuciones y productos tienen en común es el kernel de Linux. El *kernel* es al sistema operativo (SO en adelante) lo que el CPU es al hardware. El SO simplemente no puede ser entendido sin el kernel, el centro y principal componente de toma de decisiones. Cada importante elección relativa a la eficiencia, todas las Llamadas al Sistema y la administración interna de los archivos, etc. Prácticamente todo lo de relevancia termina en el kernel y va de vuelta al usuario final. Es más, junto con el firmware es el software más cercano "*al metal*", con lo cual un conocimiento profundo del propio Hardware es requisito en muchas ocasiones. Esto es particularmente cierto cuando se administran drivers, procesadores o la conexión a la red. Se trata de un tema de mucha complejidad y es difícil como ingeniero no apreciar su genialidad. Es importante recalcar que el kernel de Linux es considerado un estándar en la investigación en sistemas operativos [2], [5], [6] y en su desarrollo, tanto en cuestiones teóricas como prácticas. Esto no es casualidad, ya que se trata de un sistema hasta cierto punto modular, reflexionado y cuidadosamente revisado. Igualmente, las lecciones aprendidas pueden fácilmente ser aplicadas a otros sistemas tales como el kernel de Android o el XNU Apple kernel. El segundo derivó de FreeBSD, que está fuertemente basado en UNIX, tal y como Linux. De igual modo, el kernel de Android es una reformulación de la creación de Linus Torvalds con unos años de desarrollo por parte de Google -y la compañía que adquirieron- en una dirección distinta.

Por supuesto, tratándose el kernel de Linux de un componete tan importante del SO y en consecuencia de la eficiencia de todo el sistema en su conjunto, no es una sorpresa que una cantidad significativa de trabajo e investigación relacionados hayan sido durante años conducidos. Prácticamente cada línea de código ha sido rigurosamente revisada por expertos y muchos nuevos features junto con arreglos son periódicamente incluidos por cada nueva versión que se lanza. Sin embargo, aún queda mucho espacio para mejoras. Cada vez que un procesador es actualizado, nuevas formas de aprovechar sus características aparecen en el kernel. De igual forma, cada vez que una optimización teórica del kernel es discutida en investigaciones científicas, nuevos procesadores que la permitan deben ser construidas. De esta manera, ambas disciplinas continúan impulsando la una a la otra. Por ejemplo, Intel recientemente lanzó al mercado procesadores con tecnologías que soportaban monitorización del último nivel de cache LLC, recogidas todas en Intel RDT (Intel® Resource Director Technology) [8]–[11], permitiendo así a los desarrolladores distribuir la memoria entre hilos de forma más fructífera.

Mucho dinero y cerebro es invertido cada año para mejorar la eficiencia del CPU. Sin embargo, como con cualquier otro sistema o situación, **cómo se ejecuten las tareas es tan importante como qué tareas sean estas**. Cuando se buscan formas de mejorar la eficiencia y optimizar la producción, el orden importa. Cuando se debe controlar a varios

procesos compartiendo un mismo recursos (p.ej. el CPU o la memoria) diferentes fórmulas conocidas como algoritmos de planificación son utilizadas para discernir qué tarea ejecutar después, o para un momento dado. Los algoritmos de planificación son incluidos en el kernel como parte del *kernel scheduler* (planificador del kernel). Una mirada más cuidadosa del planificador se realiza en el Capítulo destinado al kernel.

Muchos **algoritmos de planificación** han sido propuestos a lo largo de los años. La comunidad científica ha sido testigo en la última década de un gran progreso en el diseño y desarrollo de algoritmos de este tipo [3], [6]. El objetivo principal perseguido era la justicia (*fairness*) [2], [3], [6], aportada por diseños modulares y eficientes, o incluso la garantía de algunas restricciones temporales real-time en sistemas críticos. No obstante, incluso con tanto progreso logrado de forma teórica, **el desarrollo de sistemas que implementen de verdad estos algoritmos es aún un reto**. Esto es debido a las dificultades que surgen con la implementación, que sufre de inherentes limitaciones. El kernel Linux, SMP y *expropiativo*, es un sistema complejo que requiere de un profundo nivel de comprensión de su flujo de trabajo y estructuras, más aún cuando se deben administrar varios procesadores e hiper-threading, interrupciones o careciendo de las librerías de C. En lugar de exponer un subconjunto de la ISA, todas las instrucciones están disponibles. De igual manera, la cantidad de registros útiles es mayor que en el *user space*. Los desarrolladores se ven en ocasiones forzados a emplear instrucciones de ensamblador directamente, para tareas muy específicas que son dependientes de la arquitectura. Depurar el kernel de Linux es una tarea costosa, y recompilar para cada cambio puede ser extenuante [13]. Muchas veces, un nuevo kernel es compilado e iniciado solo para toparse con un *kernel panic* (el equivalente en Windows es la "*Pantalla Azul de la Muerte*"). Huelga decir que, pocos o ningún archivo con información relevante del error pueden encontrase tras reiniciar de manera segura. Algunas partes del kernel, como el planificador de tareas, simplemente no fueron concebidas para ser modificadas mucho, lo que complica el trabajo de los desarrolladores.

9.2 Objetivos del Proyecto

Todo esto conduce a los objetivos principales de este Trabajo de Fin de Grado, relacionados con las tendencias de mercado y los dos retos técnicos descritos en las anteriores secciones. Se han identificado dos importantes problemas que podrían ser resueltos en el campo de los Sistemas:

1. El diseño, depuración y **prototipado de nuevos algoritmos de planificación** es un enorme reto incluso para desarrolladores de Sistemas especializados. Hasta el momento, esto no se hacía dinámicamente (es decir en tiempo de ejecución sin recompilar el kernel), complicando aún más el proceso. Las limitaciones de soluciones en espacio de usuario y sus restricciones accediendo a herramientas de monitorización hardware no han sino entorpecido el desarrollo de nuevos algoritmos de planificación y hecho de los framework a nivel kernel un futuro probable en el desarrollo de sistemas operativos [6].

2. Las arquitecturas más modernas plantean un nuevo reto. La investigación académica en sistemas multicore punteros probó que existe un severo degradado en rendimiento relacionado con la contención de recursos en memoria [14]. Esto es debido a que los

hilos son forzados a competir por los recursos, un aspecto que los actuales algoritmos del kernel no tienen del todo en cuenta [15], [16].

De manera acorde, **este proyecto divergió en dos direcciones** que pretenden abordar ambos retos:

1. Primero, se desarrolló una nueva versión del kernel de Linux que ayuda al diseño general y prototipado de algoritmos de planificación, FSAC (*Framework para la Creación de Algoritmos de Planificación*). Esto podría hacerse en tiempo de ejecución, evitando perder horas en recompilación del kernel y permitiendo el parcheo dinámico de los planificadores sin necesidad de reiniciar el sistema. Su desarrollo se basó en un proyecto previo LITMUS-RT [5], que si bien no contaba con prototipado dinámico, añadía ya algunas abstracciones útiles.

   Este proyecto se encuentra en fase de desarrollo -y seguramente aún necesite algún tiempo- pero a día de hoy ya se han conseguido terminar las principales abstracciones. En concreto, se han escrito miles de líneas de código del kernel, diecinueve archivos del kernel han sido modificados y creados nuevos otros catorce.

2. En segundo lugar, se ha desarrollado otro framework dentro de la herramienta open-source PMCTrack [7], que permite el prototipado de algoritmos de planificación conscientes de la situación en memoria. Se ha decidido llamarlo **PMCSched**. PMCTrack contaba ya con todo el soporte requerido para la monitorización relativa a los eventos de memoria, haciendo uso de las herramientas Intel-RDT de monitorización y distribución avanzada de recursos hardware, así como de otra información de monitorización hardware (como la consumición energética). Este parche resultó en aproximadamente tres mil líneas de código.

Sumando ambos con parches a proyectos relacionados, podemos concluir que este Trabajo de Fin de Grado requirió más de **seis mil líneas de código del kernel**. Es importante resaltar que, mientras FSAC es una nueva versión del kernel para planificado genérico, PMCSched es un parche que aprovecha un refinado proyecto anterior. Gracias a todo el trabajo previo de PMCTrack, se pudo finalizar la segunda estructura. La nueva versión ofrece soporte para algoritmos de planificación que valoran la memoria empleando herramientas hardware de monitorización del cache y del consumo de ancho de banda en el bus de memoria.

PMCSched y el kernel FSAC son ambos frameworks a nivel de SO para el prototipado de algoritmos de planificación, pero **tienen muy diferentes implementaciones, ventajas y propósito**. Mientras FSAC ofrece un prototipado de grano fino de eficientes implementaciones -a expensas de un mayor esfuerzo de desarrollo- PMCSched está optimizado para el planificado grupal y la monitorización de eventos hardware. FSAC ofrece control absoluto sobre las colas de los cores y flexibilidad, pero PMCSched puede aprovechar el resto de utilidades incluidas en PMCTrack, tales como la ejecución selectiva de hilos o la recolección de información del sistema en tiempo de ejecución.

Tanto PMCSched como el kernel de FSAC son ambos frameworks a nivel de SO de prototipado rápido de algoritmos de planificación. Dicho esto, es extremadamente importante tener en mente que tuvieron implementaciones muy diferentes y además no comparten propósito ni ven-

tajas. Mientras el kernel FSAC ofrece un preciso control "de grano fino" para imlementaciones eficientes planificadores -a expensas de una mayor dificultad en el desarrollo- PMCSched ha sido optimizado para el planificado grupal y emplear información de los contadores hardware monitorizados. Con FSAC el algoritmo tiene control absoluto sobre las run-queue de los cores y mayor flexibilidad, aunque PMCSched puede aprovechar las diversas utilidades de PMCTrack, tales como la ejecución selectiva de procesos o la información del sistema recogida por los módulos de monitorización en tiempo de ejecución.

9.3 Plan de Trabajo

En relación con el Plan de Trabajo de este proyecto, se basó en dividir los esfuerzos entre ambos proyectos. Debíamos plantearnos qué queríamos conseguir y dividir nuestros esfuerzos en metas menores, cuyo progreso pudiera ser monitorizado con una herramienta de colaboración como Trello. Las principales etapas que se fijaron fueron las siguientes.

- Se decidió dedicar un tiempo al estudio de trabajos previos. Esto no debía durar más de tres de meses, habiendo acabado por diciembre del 2019. Esto supuso revisar el proyecto LITMUS-RT y sus publicaciones y artículos asociados, así como la extensa documentación del kernel de Linux.

- Tras ello, debíamos ser capaces de ponernos manos a la obra. En lo referente a FSAC, se decidió comenzar por las cabeceras, lo que supuso otro mes de trabajo. Se decidió que para finales de marzo del 2020 debía estar la infraestructura terminada.

- Mientras tanto, se debía empezar a desarrollar la API para los plugins de PMCSched. Planeamos empezar a implementar la clase de planificación FSAC ya entrado el segundo semestre, al tiempo que se podría comenzar a dedicar tiempo a escribir algún algoritmo de prueba para PMCSched que debía estar terminado para abril del 2020. Por suerte, no solo dio tiempo a implementar un algoritmo Round Robin (RR) de grano grueso y otro consciente de memoria [17] mediante Intel-RDT como a priori había decidido, sino que también se pudo terminar un algoritmo de co-planificado grupal.

- Como mínimo, otros dos meses (abril y mayo del 2020) debían dejarse disponibles para corregir potenciales BUGs que podían aparecer tanto en la implementación de FSAC como en la nueva herramienta PMCSched.

- Finalmente, el tiempo restante podría invertirse en realizar experimentos para probar el nuevo framework de PMCTrack, publicar el código de FSAC en GitHub y redactar la memoria.

Se pudo seguir este Plan de Trabajo al pie de la letra, salvo por ajustes menores fruto de la situación del Covid-19. Durante la etapa de conducir experimentos, se escribió un script que genera reportes automáticos del comportamiento del planificador en formato .prv [18]. Estos archivos pudieron luego ser usados para crear gráficas usando un software del Barcelona Supercomputing Center.

9.4 Estructura de la Memoria

Para finalizar esta sección, sería útil recapitular guiando al lector y explicando la forma en que los contenidos de esta memoria de TFG han sido estructurados. El resto de este documento se organiza en capítulos tal y como se detalla a continuación.

- Capítulo 2 *An overview of the Linux kernel scheduler*: Un breve repaso tanto del kernel de Linux como de su planificador puede encontrarse aquí.
- Capítulo 3 *Prior work and LITMUS-RT patches*: Incluye trabajo previo relativo a lo que se pretendía conseguir, junto con una contribución a un trabajo académico relacionado.
- Capítulo 4 *Development of the FSAC kernel*: En este Capítulo se resume el desarrollo de la API del kernel FSAC. Antes, un borrador de un hipotético algoritmo de planificación plugin FCFS se desarrolla, con el objetivo de determinar qué debería ofrecer la API de FSAC, al menos en su versión más básica y rudimentaria. Después se detallan las modificaciones internas requeridas en el kernel de Linux, en especial para el soporte real-time.
- Capítulo 5 *PMCSched: A PMCTrack-based scheduling framework*: Aquí se describe el desarrollo de la API para el framework de planificado de PMCTrack PMCSched y la construcción de este acto seguido.
- Capítulo 6 *Contention-aware scheduling algorithm*: Una explicación de cómo programar un algoritmo de co-planificado [19] de n-cores y otro consciente de memoria [17] con PMCTrack se detalla aquí.
- Capítulo 7 *Experimental Evaluation*: En este Capítulo se discuten los resultados experimentales obtenidos con el nuevo algoritmo consciente de Memoria y se comparan con los del planificador actual de Linux.
- Capítulo 8 *Conclusions and Future Work*: Este es el capítulo de conclusiones donde también se habla de posible trabajo futuro.
- Apéndice A *Installation and replication guide*: En esta sección se discute el entorno de trabajo y metodología, sin entrar en demasiados detalles, como forma de empezar a sumergirme en este proyecto y hacer posible la replicación.
- Traducción al español de varias partes de la memoria, incluidos los capítulos 1 y 8.

Chapter 10

Conclusiones y Trabajo Futuro

Una multitud de nuevas tecnologías tales como el Big Data o la aceleración del Machine Learning han puesto al hardware en jaque. Sorprendentemente, por primera vez en muchos años el hardware está perdiendo la carrera a las demandas del mercado. Intentar enfrentarse a estos nuevas aplicaciones es habitualmente muy exigente a nivel computacional, o directamente imposible para el ordenador que uno podría esperarse encontrar en cualquier hogar. Como resultado de todo esto, solo las mayores compañías tecnológicas pueden permitirse recolectar, analizar y extraer por sus propios medios algo útil de toda esa información.

Al mismo tiempo, mejorar el hardware se ha ido complicando más y más, al menos a la velocidad en que la industria de los semiconductores estaba habituada. De hecho, sólo unas pocas compañías aún intentan cumplir la profecía de Moore, aunque estén afrontando retos de creciente complejidad. Por esto algunos investigadores comenzaron a referirse a este periodo como la *Era Post-Moore* [1].

Pero no debemos perder toda esperanza. En la capa superior (el sistema operativo) muchos componentes no han sido apenas actualizados por décadas, relegando en la mejoría del hardware o de capas superiores del *software stack*. Existen incontables posibilidades de mejora en esta dirección.

El principal problema es que crear y optimizar software al nivel del SO no es para nada sencillo. Es necesario tener un conocimiento profundo tanto del hardware que se va a administrar como de las aplicaciones que intentarán usarlo. Por ejemplo, uno no puede escribir el driver para un dron si no está familiarizado -como mínimo- con el firmware específico de su microcontrolador y al mismo tiempo con la interfaz que el usuario empleará para enviar comandos.

Además, hay una necesidad de mejorar los algoritmos de planificación presentes en los sistemas actuales, ya que no tienen del todo en cuenta muchos factores. Uno de estos es el degradado en rendimiento que surge debido a los recursos compartidos: En sistemas multicore, las aplicaciones pueden llegar a competir por hacer uso intensivo de recursos compartidos [3], tal y como son el controlador DRAM, el último nivel de cache o el bus. A este se le suma el hecho de que esto conduzca a situaciones injustas en las que algunas aplicaciones hagan uso más intensivo que otras de dichos componentes.

Basados en estas observaciones, el objetivo principal de este proyecto fue el de facilitar el trabajo de los desarrolladores de SO que intentan mejorar un componente en concreto del kernel de Linux, el planificador. Se trata de uno de los elementos más críticos ya que está a cargo de distribuir el tiempo de CPU entre los hilos y realizar asignaciones core-a-hilo. Para cumplir esto, el proyecto del kernel FSAC (*Framework for Scheduling Algorithm Creation*) fue comenzado. Al mismo tiempo, se desarrolló PMCSched para dar soporte al prototipado de algoritmos de planificación en PMCTrack [7].

Para solucionar el problema del desarrollo en el SO, se comenzó el proyecto **FSAC**, que incluye una nueva variante del kernel de Linux para facilitar el diseño general y prototipado de algoritmos de planificación. Este framework ahorra horas de recompilación del kernel, ya que esto puede hacerse en tiempo de ejecución (*runtime*) sin necesidad de reiniciar el sistema. Su desarrolló se basó en un proyecto previo, LITMUS-RT [5], que si bien no permitía el prototipado dinámico, contaba con interesantes abstracciones. Todo el trabajo previo y las contribuciones a este proyecto se detallaron en el capítulo 3 "Prior work and LITMUS-RT patches".

En segundo lugar, otro framework llamado PMCSched se desarrolló dentro de la herramienta de monitorización open-source PMCTrack, para permitir el prototipado de algoritmos conscientes de los recursos. PMCTrack ya tenía todo el soporte necesario para monitorizar eventos de memoria, empleando las avanzadas tecnologías de monitorización de Intel [8]–[11], así como acceso a otra información hardware (como el consumo energético). Sumando ambos con parches a proyectos relacionados, podemos concluir que este proyecto involucró **más de seis mil líneas de código del kernel**.

PMCSched y el kernel FSAC son ambos frameworks a nivel de SO para el prototipado rápido de algoritmos de planificación. Dicho esto, es realmente importante recalcar que tuvieron implementaciones muy distintas, y más importante aún, que no comparten propósito o ventajas. Mientras el kernel FSAC ofrece un control muy preciso para algoritmos eficientes -a expensas de un mayor esfuerzo de desarrollo- PMCSched ha sido optimizado para el planificado grupal y algoritmos que requieran emplear información de monitorización de rendimiento. Con FSAC el algoritmo tiene pleno control sobre los run-queue de los cores y mayor flexibilidad, mientras PMCSched puede aprovechar diversas utilidades de PMCSched, como son la selección selectiva de un subconjunto de los procesos o la información del sistema recogida por los módulos de monitorización en tiempo de ejecución.

Una vez PMCSched estuvo completado, decidimos ponerlo a prueba creando un algoritmos consciente de la contención. De esta forma pudimos al mismo tiempo probar la robustez del framework y experimentar con ideas recientemente propuestas para solucionar el problema de contención de recursos compartidos.

Los resultados obtenidos con el algoritmo de planificación *MEMSCHED* desarrollado en PMCSched fueron satisfactorios. En particular, medimos una mejora significativa en el *system throughput* (STP) de hasta un 24% en comparación con el actual planificador de Linux. Casi **50 benchmarks fueron empleados en nuestros experimentos**. Con un script que desarrollamos para generar trazas de la ejecución, fuimos capaces de crear gráficas con los valores, dando estas interesante información relativa al comportamiento de los algoritmos.

10.1 Trabajo Futuro

El mismo usuario puede utilizar un ordenador para cargas de trabajo de lo más diversas. Por ejemplo, esa persona puede estar deseando jugar a videojuegos online, editar unas fotografías o quizás estar interesado en reproducir videos en streaming. Todas estas cosas se consiguen con aplicaciones que el sistema operativo debe administrar por debajo, y debe proveer con cuantos recursos sea posible.

Cada uno de los mencionados ejemplos tiene muy poco en común, y los procesos involucrados en su cumplimiento tendrán requisitos muy distintos. Presumiblemente, los videojuegos harán uso frecuente del hardware de red, mientras que el procesamiento de imagen hace uso intensivo del GPU. Algunos de ellos usarán los recursos de memoria continuamente y otros tendrán suficiente con el espacio provisto para ellos en los niveles privados de la caché del procesador.

Sin embargo, a día de hoy todos ellos recibirían el mismo trato por parte del sistema operativo, salvo que alguna restricción de prioridad fuera explícitamente especificada. Todas estas necesidades permanecen a día de hoy transparentes para el kernel, y esto indudablemente supone un impacto en el rendimiento. El kernel de Linux está presente en cada supercomputadora de la lista TOP 500, y si embargo aún contempla solo tres familias de aplicaciones: Hard Real Time, Soft Real Time y las por defecto.

Existen multitud de posibilidades para mejorar el planificador y hacerlo consciente sobre las necesidades particulares de las aplicaciones. Con un Framework como PMCSched, los profesionales IT y los científicos podrían incluso hacer uso de sistemas ad-hoc, con planificadores específicamente concebidos para sus necesidades.

Por citar un ejemplo, las compañías con alta demanda de entrenamiento de IA, podrían usar planificadores especialmente diseñados para mejorar el rendimiento en cargas de trabajo de esa naturaleza. Quizás -y esto es sólo un ejemplo hipotético- unos investigadores descubren que existen cierto tipo A de hilos de IA que es mejor que sean ejecutados simultáneamente con los de otro tipo, mientras que los hilos de entrenamiento IA de un tercer tipo es preferible que sean ejecutados en solitario, resultando todo esto en menos cache-misses, contención de ancho de banda y finalizando la ejecución antes.

En lo relativo a posibles **mejoras para este proyecto**, PMCSched y el algoritmo *MEMSCHED* podrían sin duda ser mejorados. De hecho, consideramos que el kernel *FSAC* ha sido desarrollado de manera suficientemente modular como para seguir mejorándolo en un futuro sin demasiado esfuerzo. El nuevo feature que podría ser más de más inmediata incorporación a FSAC, en este punto, sería un conjunto de scripts *wrapper* para interactuar con la interfaz basada en `/proc`.

Para la extensión de PMCTrack, PMCSched podría hacer incluso más por el desarrollador. Hasta ahora, la API de desarrollo de algoritmos de planificación no provee de abstracciones para el particionado de cache con las tecnologías Intel CAT [12]. Así mismo, la frecuencia en que se invoca a la función de planificado periódica fue establecida arbitrariamente, cuando otros valores podrían quizás arrojar mejores resultados en cuanto a rendimiento o justicia.

Otra potencial mejoría sería la de añadir soporte para la obtención automática de la información

del sistema, tal y como el máximo ancho de banda soportado, en lugar de añadir esos valores a mano en el código fuente o pedir al desarrollador que los configure.

El algoritmo consciente de memoria *MEMSCHED* podría ser mejorado también. El umbral establecido para diferenciar aplicaciones y categorizarlas en función de su comportamiento para con la memoria fue también establecido de manera arbitraria en cierta medida: Se fijó tras revisar los resultados obtenidos por varios benchmark con un comportamiento ya conocido con antelación. No obstante, el algoritmo podría sin lugar a dudas verse beneficiado por unos valores más precisos del umbral. Así mismo, la complejidad del algoritmo podría mejorar de $\mathcal{O}(n \log n)$ a $\mathcal{O}(kn)$ si se mejorara la manera en que se administra la lista doblemente enlazada. Por último, también sería interesante considerar que el *profiling* de los hilos -explicado en el capítulo 6- aumenta la injusticia. Esto es así porque, mientras el primer hilo a administrar debe ser detenido por cada nuevo, el último nunca tendrá que esperar lo que dure la fase de profiling de otro.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.

[2] A. Garcia-Garcia, J. C. Saez, and M. Prieto-Matias, "Contention-aware fair scheduling for asymmetric single-isa multicore systems," *IEEE Transactions on Computers*, pp. 1–1, 2018.

[3] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, vol. 45, no. 1, Dec. 2012.

[4] ARM, "Benefits of the big.LITTLE Architecture." [Online] Available: http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf; ARM.

[5] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," PhD thesis, The University of North Carolina at Chapel Hill, USA, 2011.

[6] A. Garcia-Garcia, J. C. Saez, F. Castro, and M. Prieto-Matias, "LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores," in *Proceedings of the 48th international conference on parallel processing*, 2019.

[7] J.C. Saez and J. Casas and A. Serrano and R. Rodriguez-Rodriguez and F. Castro and D. Chaver and M. Prieto-Matias, "An OS-oriented performance monitoring tool for multicore systems," in *Euro-par 2015: Parallel processing workshops*, 2015, pp. 697–709.

[8] K. Nguyen, "Intel's cache monitoring technology software-visible interfaces." [Online] Available: https://software.intel.com/en-us/blogs/2014/12/11/intel-s-cache-monitoring-technology-software/-visible-interfaces; Intel Corporation, 2014.

[9] K. Nguyen, "Intel's cache monitoring technology: Use model and data." [Online] Available: https://software.intel.com/en-us/blogs/2014/12/11/int/els-cache-monitoring-technology-use-models-and-data; Intel Corporation, 2014.

[10] Intel Corporation, "Intel® 64 and ia-32 architectures developer's manual: Vol. 3B." [Online] Available: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html; Intel Corporation, 2014.

[11] K. Nguyen, "Benefits of intel(R) cache monitoring technology in the intel(R) xeon(TM) processor e5 v3 family." [Online] Available: https://software.intel.com/en-us/blogs/2014/06/18/benefit-of-cache-monitoring; Intel Corporation, 2014.

[12] K. Nguyen, "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family." [Online] Available: https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology; Intel Corporation, 2016.

[13] R. Love, *Linux kernel development, 3rd edition*. Addison-Wesley Professional, 2010.

[14] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with intel's cache allocation technology," in *2017 26th international conference on parallel architectures and compilation techniques (pact)*, 2017, pp. 194–205.

[15] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*, 2012, pp. 83:1–83:11.

[16] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Cache Contention in Multicore Processors Via Scheduling," in *15th int'l conf. Architectural support programming lang. And oper. Syst. (ASPLOS 10)*, 2010, pp. 129–142.

[17] T. Marinakis and I. Anagnostopoulos, "Performance and fairness improvement on cmps considering bandwidth and cache utilization," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 1–4, 2019.

[18] Barcelona Supercomputing Centre, "Paraver: A flexible performance analysis tool." [Online] https://tools.bsc.es/paraver, 2018.

[19] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Addressing fairness in smt multicores with a progress-aware scheduler," in *2015 ieee international parallel and distributed processing symposium*, 2015, pp. 187–196.

[20] W. Mauerer, *Professional linux kernel architecture*. Birmingham,UK,UK: Wrox Press Ltd., 2008.

[21] J. Aas, *Understanding the linux 2.6.8.1 cpu scheduler*. Silicon Graphics, Inc (SGI), 2011.

[22] M. A. Dahl G and T. Newhall, "Parallelizing neural network training for cluster systems," 2008.

[23] "LITMUS-rt website." [Online]. Available: http://www.litmus-rt.org/.

[24] "Linux kernel mailing list archive." [Online]. Available: https://lkml.org/.

[25] "4.9.30 linux stable release with pgp signature." [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v4.9.30.

[26] J. C. Saez, A. Pousa, R. Rodriíguez-Rodriíguez, F. Castro, and M. Prieto-Matias, "PMCTrack: Delivering performance monitoring counter support to the os scheduler," *The Computer Journal*, vol. 60, no. 1, pp. 60–85, 2017.

[27] A. Garcia-Garcia, J. C. Saez, J. L. Risco-Martin, and M. Prieto-Matias, "PBBCache: An open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and

cache-clustering policies," *Journal of Computational Science*, vol. 42, p. 101102, 2020.

[28] J. C. Sáez, J. I. Gomez, and M. Prieto, "Improving priority enforcement via non-work-conserving scheduling," in *2008 37th international conference on parallel processing*, 2008, pp. 99–106.

[29] B. Gregg, "Linux extended bpf (eBPF) tracing tools." [Online]. Available: http://www.brendangregg.com/ebpf.html.