POLITECNICO DI TORINO

Master's Degree Thesis in Computer Engineering

Master's Degree Thesis

Hardware-based schedulers approaches for Linux OS

Supervisors
Prof. Alessandro SAVINO
Prof. Maurizio REBAUDENGO

Candidate
Leonardo IZZI

October 2021

Abstract

Tasks scheduling is one of the most critical activities in any operating system. Choosing which task should possess a CPU core (and for how long) heavily influences the system's performance. A common approach for solving these problems in today's scheduling algorithms is to consider the task's priority and execution time of all processes. However, modern hardware optimization structures, such as hierarchical caches and branch prediction units, influence the execution time, leading to suboptimal results if misused. Even though such information is made available to software through Performance Monitoring Counters (PMCs), no known scheduler uses them to improve scheduling decisions. This work integrates PMC-based evaluations in the Linux Completely Fair Scheduler (CFS) to study how thread scheduling may benefit from such knowledge. Results confirm the broad fluctuations in task turnaround time depending on the monitored structures, providing insights into possible CFS optimization.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Hardware systems have grown significantly in complexity to cope with the ever-increasing demand of today's applications. Modern microprocessors can run billions of instructions per second on multiple cores, while software programs must coordinate a multitude of parallel tasks. Operating Systems (OSes) are key to manage such complexity: they abstract the hardware platform, manage the resources on behalf of user programs and offer a set of APIs to ease application development and communicate with the OS itself.

The thread scheduler is a critical component present in any operating system. Its main purposes are to select which task should possess the CPU and for how long [40], guaranteeing fairness among tasks while respecting priorities. In a multi-core environment additional complications arise for the scheduler, as load balancing algorithms [8] are necessary to even the work across the execution units.

The Linux Completely Fair Scheduler (CFS), the default scheduler for standard tasks, selects the task which has spent less time on the CPU as the next one to run, and it assigns the execution quantum as a proportion of the scheduling period [30]. This information, however, is insufficient to characterize a thread's behavior on modern hardware. Today's CPU offer many mechanisms to speed-up code execution, such as branch predictors, hierarchical caches, speculation, out-of-order execution, and so on. Their effectiveness strongly depends on how code interact with the underlying hardware.

Performance Monitoring Counters (PMCs) are tools embedded in virtually any CPU that *counts* the occurrences of microarchitectural events. They are widely used by developers of user-space programs to identify and solve performance bottlenecks in their code. In fact, there exist many applications and libraries to interact with them from userland, such as *perf* [35], *PAPI* [41] and *PMCTrack* [38]. Although PMCs provide useful insight for characterizing tasks' behavior, no known scheduler make use of such knowledge to improve its decision process. Moreover, there are fewer solutions to access performance counters from within the kernel, and they all have shortcomings: *perf*,

the built-in Linux performance counters subsystem, introduces a non-negligible overhead which hurts performance in a critical component like the scheduler [44]. PMC-Track, on the other hand, due to its module structure does not allow a strong integration with the scheduler.

Hence, this thesis focuses on the development of a lightweight PMC library and its introduction in the Linux CFS to alter scheduling decisions with PMCs-based evaluations. The remaining of this document is organized as follows: chapter 2 is an introduction to performance counters and their structure on Intel platforms. Chapter 3 presents the process/thread structure in the Linux kernel and discusses the internals of the Linux scheduler, focusing in particular on CFS operations. Chapter 4 is the bulk of this work and explain the structure of the PMC library developed for the scheduler, as well as the methodology used to build PMC-scheduling within the CFS. Chapter 5 describes the test architecture developed for benchmarking and analyzes the experimental results obtained from the tests. Finally, chapter 6 presents a summary of the work carried in this thesis and discusses the possible future works.

Chapter 2

Performance Monitoring Counters

2.1 PMCs introduction

Performance monitoring counters are tools used for software performance analysis and tuning. They are a set of registers whose purpose is to store the count of hardware-related events inside a microprocessor. Usually, the number of events that can be monitored far exceeds the number of available registers. For this reason, PMCs are paired with configuration registers to select the event to be monitored and fine-tune the counter behavior.

In multi-core and multi-thread environments, every execution unit has its own private copy of the register set. This means that each thread is responsible for the configuration of its performance counters, but also that the monitor process is local to hardware threads, allowing users to obtain precise information. However, for shared resources, the situation is more complicated. The Last Level Cache (LLC), for example, is often divided by all cores in the package. For these components there exist two main counting approaches: in the first one, each core counts only the events generated by itself, while in the second one there is a "global" register counting the events arising from every thread.

These solutions have both their advantages and disadvantages. With dedicated counters it is possible to know exactly how a particular thread interacted with the resource, on the other hand it can be hard to monitor the global behavior. The converse is true for the global resource counter.

2.2 Intel architecture

2.2.1 Model-Specific Registers

On Intel architectures, performance monitoring counters are a subset of Model Specific Registers (MSRs) [22]. MSRs have many possible uses, such as getting detailed system information, CPU resources configuration and the management of the processor's features.

MSRs can be broadly divided in two main categories: *architectural* and *non-architectural*. The former are guaranteed to not change in future processors, while the latter may be changed, removed or replaced from one micro-architecture to the other. However, even architectural MSRs or bit fields may be introduced or deprecated between processor families. Accessing a non-existent MSR generates an exception, whilst configuring it in the wrong way leads to unexpected results.

To distinguish processors and obtain feature information, the x86 and x86-64 architectures support the `cpuid` instruction. Based on the values contained in the `EAX` and (optionally) `ECX` registers, this instruction returns the aforementioned details in `EAX`, `EBX`, `ECX` and `EDX`.

MSRs are accessed through special instructions, `RDMSR` and `WRMSR`. `RDMSR` reads the register value, `WRMSR` overwrites it. Both instructions use `ECX` as source address. Each MSR is 64 bits long, therefore on x86 `EDX` and `EAX` are needed to store the value to be read or written. `EDX` holds the higher part, `AEX` the lower one. This is true also for x86-64 for backward compatibility reasons.

Model-specific registers are often unaccessible from user space due to security reasons. Hence, the kernel or a kernel module must provide a software APIs to interact with them from user-space.

2.2.2 Architectural Performance Counters

Performance facilities were initially added as non-architectural MSRs in the first Pentium. Starting with Intel Core Solo and Intel Core Duo processors, the monitoring system became architectural with support for both architectural and non-architectural events.

There exist multiple architectural performance monitoring versions, that can be identified by the bits 7:0 of the `EAX` register when `cpuid` is called with `EAX` set to 0x0A. A higher version ID corresponds to a newer version, where more features and capabilities are available. The first version defines the MSRs used for configuration and counting, their addresses, bit fields and bit widths. These registers are called IA32_PERFVTSELx and IA32_PMCx respectively, with x being the ID (0, 1, 2, 3...) [23]. They occupy a contiguous block of MSR addresses which starts at 0x186 for IA32_PERFVTSELx and at 0xC1 for IA32_PMCx. Every core has eight PMCs units available, but processors with Simultaneous MultiThreading (SMT) enabled are allowed to use only four PMCs per thread. The bit fields of IA32_PERFVTSELx is shown in figure 2.1, where:

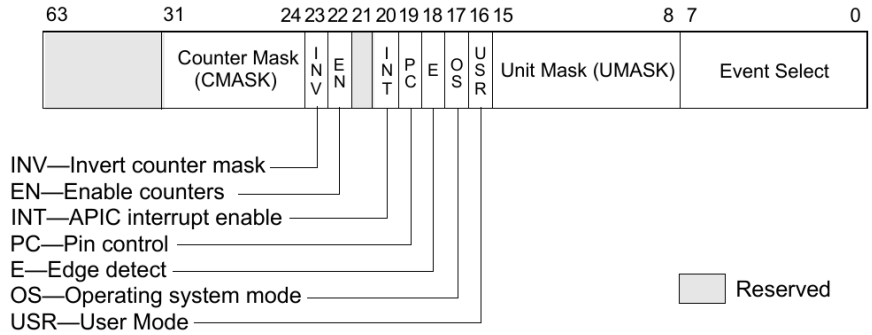


Figure 2.1: IA32_PERFVTSELx version 1 layout. Taken from [23].

- **Event Select (bits 7:0)**: Select the event logic unit used to detect microarchitectural events.
- **Unit mask (UMASK) (bits 15:8)**: Select the event to be monitored within the event logic unit.
- **USR (user mode) (bit 16)**: Determines if the events must be counted in privilege levels other than 0.
- **OS (operating system mode) (bit 17)**: Determines if the events must be counted in privilege level 0.
- **E (edge detect) (bit 18)**: If enabled the counter increases its value when the event signal executes a low-to-high transition.
- **PC (pin control) (bit 19)**: When set, enables the toggling of the PMi pins when an event occurs. It is reserved since the Sandy Bridge architecture.
- **INT (APIC interrupt enable) (bit 20)**: When set, generates an interrupt when the counter overflows.
- **EN (Enable counters) (bit 22)**: When set, it enables the corresponding IA32_-PMC.
- **INV (invert) (bit 23)**: When set, inverts the CMASK comparison.
- **Counter mask (CMASK) (bits 31:24)**: The counter compares this value against the number of occurred events in a single clock cycle. If the number of events is greater or equal to the mask, the counter value is increased.

The IA32_PERFVTSELx registers support both architectural and non-architectural performance events. An architectural event is guaranteed to have always the same

Event Mask Name	Event Number	UMASK Value
Unhalted Core Cycles	0x3C	0x00
Unhalted Reference Cycles	0x3C	0x01
Instructions Retired	0xC0	0x00
LLC Reference	0x2E	0x4F
LLC Misses	0x2E	0x41
Branch Instruction Retired	0xC4	0x00
Branch Misses Retired	0xC5	0x00

Table 2.1: Architectural performance events

functionality and configuration values. A programmer which does not want to distinguish between processors families should only rely on such events to monitor performance. However, the event list is rather short, as shown in table 2.1.

The architectural events are:

- **Unhalted Core Cycles**: Increases the counter with core frequency when it is in power state C0. The number of counted events per unit time changes with state transitions, although during such transitions the counting process is stopped. A state transition corresponds to a frequency scaling.
- **Unhalted Reference Cycles**: Increases the counter at a fixed frequency when the core is in state C0. Depending on the processor, different clock sources may be used, each providing a different method to obtain the reference frequency value.
- **Instructions Retired**: Counts the number of instructions retired. In case of a multi-μop instruction, the counter increases its value only when the last μop has successfully concluded its execution.
- **LLC Reference**: Counts the number of memory operations reaching the LLC. Depending on the processor, the LLC may correspond to the L2 or L3 cache. This counter also includes events generated by different sources other than the core itself, such as the hardware prefetcher.
- **LLC Misses**: Counts the number of memory operations generating a miss in the LLC. As for the *LLC Reference* counter, the LLC may correspond to different cache levels and any source generating LLC misses is considered in the counting process.
- **Branch Instruction Retired**: Counts the number of branch instructions retired. If a branch instruction translates to a multi-μop operation, the counter is increased only when the last μop is retired.
- **Branch Misses Retired**: Counts the number of mispredicted branches. If a branch instruction translates to a multi-μop operation, the counter is increased only when the last μop is retired.

2.2.3 Haswell Performance Events

Although architectural events are the perfect choice for cross-generation compatibility, they are insufficient to fully characterize the behavior of a program executing on the CPU. For this reason, the set of performance events presented in subsection 2.2.2 is extended by platform-specific options. A comprehensive events list for every micro-architecture is available on the Intel manual [23] and on Intel website [21], which is also updated in case of hardware errata.

Table 2.2 presents the relevant model-specific events used in this work.

Event Mask Name	Event Number	UMASK Value
MEM_LOAD_UOPS_RETIRED.L1_HIT	0xD1	0x01
MEM_LOAD_UOPS_RETIRED.L2_HIT	0xD1	0x02
MEM_LOAD_UOPS_RETIRED.L3_HIT	0xD1	0x04
MEM_LOAD_UOPS_RETIRED.L1_MISS	0xD1	0x08
MEM_LOAD_UOPS_RETIRED.L2_MISS	0xD1	0x10
MEM_LOAD_UOPS_RETIRED.L3_MISS	0xD1	0x20
MEM_LOAD_UOPS_RETIRED.HIT_LFB	0xD1	0x40

Table 2.2: Haswell performance events

The MEM_LOAD_UOPS_RETIRED.Lx_HIT event, with x being the cache level, increases the counter value whenever a load μop produces a hit in the corresponding cache. MEM_LOAD_UOPS_RETIRED.Lx_MISS, instead, increases the counter value whenever a load μop produces a miss in the related cache.

The L1 cache events have a slightly different behavior when compared to their counterparts in the other cache levels. The cache hit counter increases its value by 1 irrespectively of the load size. On the other hand, the cache miss counter increases its value by 1 only the first time a miss is produced on the cache line. Any other load producing a cache miss on the same cache line will increment the value of the MEM_LOAD_-UOPS_RETIRED.HIT_LFB event, which is the number of cache misses in L1 that have produced a hit in the Line Fill Buffer.

It is important to remember that, a programmer wishing using these events, must either use the `cpuid` instruction to recognize the processor family and model, or is sure that the code will only run on the selected architecture.

Chapter 3

Linux Thread Scheduling

3.1 Basic Concepts

3.1.1 Programs, Processes and Threads

A *process* is defined as a running instance of a *program* [40]. Usually, a program is a file stored in mass memory containing the code, data and other information used by the program loader and linker. In the Unix world, the de-facto standard is the ELF file format [5].

When a program starts, it is loaded in memory by the OS. In a first approximation, the memory layout is the one depicted in figure 3.1.

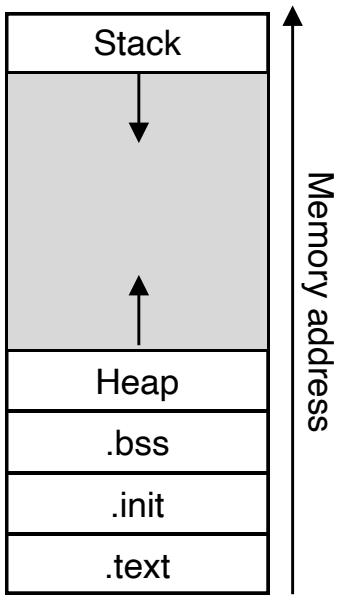


Figure 3.1: Process memory layout

The `.text` section stores the program's code, while the `.init` holds the initialized global variables. The `.bss` is also a data section, but it stores the uninitialized global variables. For this reason, a common optimization is to save only the section's length on disk to reduce the executable size. Depending on the operating system, the `.bss` section may be 0-filled when loaded in memory [31], hence the global variables initialized to 0 may be stored in the `.bss` section as well, sparing even more space.

The *stack*, instead, is a dynamic memory portion where local variables are saved. In Linux, the stack size has a hard-coded size limit set by default to two pages on x86-64 [26], but it can be changed while configuring the kernel compilation. Finally, the *heap* is a dynamic memory region storing data that must escape a function's lifetime.

While this description may be enough from a user's perspective, it is not from the kernel's one. A process is a much more complicated entity, as the OS must keep track of opened files and sockets, the process' address space, state and more.

Threads represent an execution flow within a process [30]. The scheduler operates on threads, not processes. In Linux, a process has at least one thread, but is allowed to use more of them. In this case, the program is called *multi-threaded*. Each thread consists of a unique copy of the CPU registers and has a dedicated stack, everything else is shared with its process. For this reason, threads are often referred to as *lightweight processes*.

3.1.2 Task Structure

The Linux kernel does not distinguish between threads and processes: it uses the same structure to describe both of them, called `task_struct`. It is a large data structure, composed by many fields. For compactness it is not reported here, but the definition is available in `<include/linux/sched.h>`.

This structure is also called process descriptor, as it stores the process and threads information needed for scheduling, memory management, the file descriptors opened and so on.

Tasks Relationship

Linux organizes the processes in a hierarchical manner, as shown in figure 3.2. The root process is called `init` and it is the first task created by the kernel after a boot. In general, a process has exactly one parent and zero or more children. A process is also aware of its sibling, that is, the tasks sharing the same parent. These relationship are stored in the `task_struct` in the following fields:

```
1    struct task_struct __rcu *real_parent;
2    struct task_struct __rcu *parent;
3    struct list_head children;
4    struct list_head sibling;
```

`real_parent` is a pointer to the parent task's `task_struct`. However, it may happen that the parent task ends its execution earlier than expected, for example due to

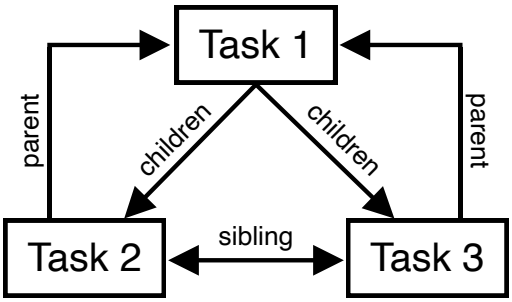


Figure 3.2: Tasks relationships

a crash or an unhandled signal. Therefore, the `task_struct` has an additional field called `parent`, which either points to the actual parent task or to `init` in case of a premature termination.

`children` is a double circular linked list, which can be iterated to obtain the children's `task_struct`. However, it is not correct to directly access this value: the kernel list implementation works by embedding the list structure within the structures that must be iterated. Hence, to obtain the `task_struct` pointers, it must be used the API defined in `include/linux/list.h`.

Finally, `sibling` is a double circular linked list as well, pointing to the `task_struct` sharing the same parent. This functionality is used, for example, to implement *cgroup* [6].

Task Identifiers

In Unix, a user space task is uniquely identified by an ID called *pid* (Process ID). The pid is defined with an opaque type called `pid_t`. In older Linux versions it would translate to a `short int`, that is to a value 16 bits wide. This means that the maximum number of concurrent processes has an hard limit of 65536.

Pids are usually assigned with increasing values, however it is legal to wrap up the count when the number does no longer fits 16 bits. It is possible to increase the maximum limit, at the cost of breaking compatibility with older programs.

The kernel defines two different fields in `task_struct` to manage task identifiers: `pid` and `tgid` (Thread Group ID), both of `pid_t` type. In a rather misleading way, the user space pid does not map to the task's `pid`, but to `tgid` [15]. Internally, the kernel uses `pid` as a unique task identifier, while `tgid` is shared by every thread in a process. The result is that for a single-thread process `pid = tgid`. On the other hand, in multi-thread process the previous equality holds only for the first thread, as new threads will obtain a different `pid` values.

Task Creation

The kernel creates a new task every time a new process or thread is spawned. As stated earlier at the beginning of section 3.1.2, the kernel uses the same structure to handle processes and threads. In fact, system calls like `fork()` or `pthread_create()` are just wrapper around `clone()` [4].

The `clone()` system call allows to fine-tune what resources of the parent should be *cloned* and what should be *copied* through the usage of a bit mask. The distinction between a clone and a copy is subtle but important: the former means that the tasks *share* the same object, the latter means that each task possess a *distinct copy* of the object.

The kernel's `clone()` declaration is:

```
long clone(unsigned long clone_flags, unsigned long newsp,
           int __user *parent_tidptr,
           int __user *child_tidptr,
           unsigned long tls);
```

In the first parameter, the user-space caller specifies the bit mask determining how the task cloning should be performed. For `fork()`, `clone_flags` is set to

```
clone_flags = SIGCHLD
```

Instead, to create a thread, it is set to

```
int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SYSVSEM
                   | CLONE_SIGHAND | CLONE_THREAD
                   | CLONE_SETTLS | CLONE_PARENT_SETTID
                   | CLONE_CHILD_CLEARTID
                   | 0);
```

Copy-on-write After a `fork()`, the newly created process is an exact copy of the parent. To actually execute a new program, the `exec()` system call family must be used. However, this means that most of the work done by `fork()` is wasted. The criticality of the problem is particularly evident when considering how much expensive is the copy of the whole address space.

To reduce the performance penalty to the minimum, the kernel exploits a copy-on-write mechanism [8]: the memory pages are not immediately duplicated, but are marked as read-only. Whenever the parent or the child writes one of them, the kernel receives an exception and duplicates the page with the correct access rights. Hence, in case of a `fork()` immediately followed by an `exec()` system call, no memory page is duplicated at all.

Clone implementation Within the kernel's code, `clone()` is a wrapper for `_do_fork()`. This function, in turn, calls `copy_process()`, which performs the bulk of the `task_struct` initialization. The job of this function can be summarized as follows:

- Checks the validity of the flags passed by `clone`.
- Duplicates the task structure and allocate a new `task_struct`, stack and the architecture-specific `thread_info`.
- Clears or initialize various resources, for example the ones used for statistics purposes.
- Assigns a new pid.
- Based on `clone`'s flags, resources such as file descriptors, memory space structures, etc. are either shared or copied.

Once `copy_process()` returns to `_do_fork()`, if the task creation succeeded the new child is woken up and put in a runqueue (refer to section 3.2.3 for more informations about runqueues).

Task States

During its lifetime, a task may switch among different states, depicted in figure 3.3. The kernel encodes this information in the `task_struct`'s `state` bit mask. The main values assumed by this field are:

- `TASK_RUNNING (0x0000)`: the task is either ready to run or is running on the CPU.
- `TASK_INTERRUPTIBLE (0x0001)`: The task is blocked, waiting for a condition to happen. It is possible for a signal to awake the task, even if the condition has not occurred yet.
- `TASK_UNINTERRUPTIBLE (0x0002)`: It is the same as `TASK_INTERRUPTIBLE`, but a signal cannot prematurely wake the task.
- `__TASK_STOPPED (0x0004)`: The task is stopped, that is, it cannot resume its execution. Such state is entered when the `SIGSTOP`, `SIGSTP`, `SIGTTIN` signals are received by the task, or when any signal is received while it is being debugged.
- `__TASK_TRACED (0x0008)`: The task is being traced by another application, such as a debugger, through `ptrace`.

Task Termination

Every task, sooner or later, must end its execution. From user space, processes and threads use different mechanisms to communicate their termination to the kernel. In particular, a process uses `exit()`, while a POSIX thread calls `pthread_exit()`. However, it is also possible for a task to be forcibly ended as a result of a system exception,

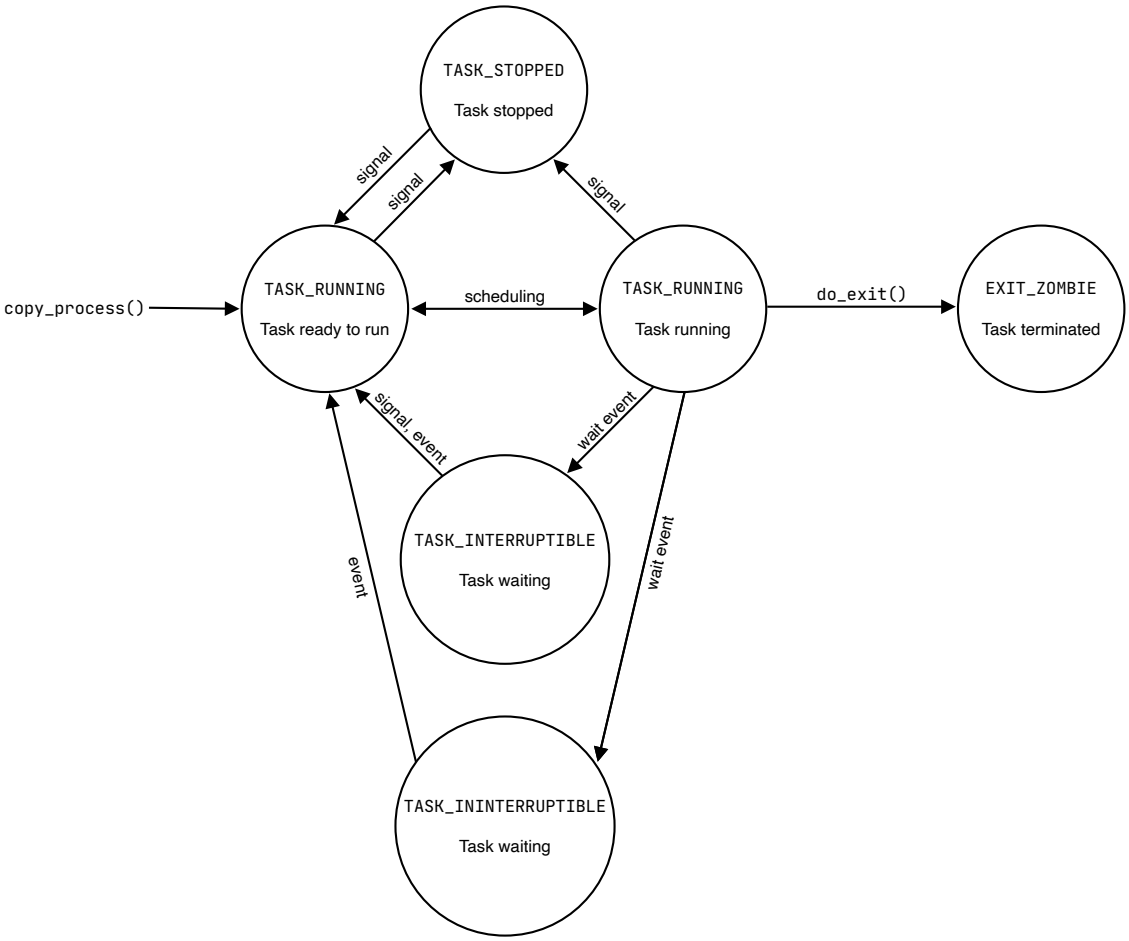


Figure 3.3: Tasks states diagram

an unhandled signal or an un-maskable signal, triggered for example as a response to a null pointer access.

The kernel uses a different bit mask than `state` to indicate the exit state, called `exit_state`. The valid values are:

- `EXIT_DEAD (0x0010)`
- `EXIT_ZOMBIE (0x0020)`
- `EXIT_TRACE (EXIT_DEAD | EXIT_ZOMBIE)`

`EXIT_ZOMBIE` is a special state assumed by the task in the time between the end of its execution and the parent read of the exit value. If the parent dies before its children, they remain in this state until the kernel is rebooted.

Under the hood, when a process or a thread terminates, the C standard library redirects the execution flow to the kernel `do_exit_group()` and `do_exit()` functions respectively. `do_group_exit()` sends a signal to each thread, if any, to terminate them as well, then it calls `do_exit()`. `do_exit()` is the main exit function, where most of final processing and resources are cleaned up. At a high level, this function executes these steps:

- Sets a task flag to `PF_EXITING`.
- Cancels any timer created by the task.
- Releases the memory assigned to the task.
- Closes (and possibly frees) the file descriptors and filesystem data.
- Removes the task from any scheduling-related structure.
- Notifies the parent of the child death sending the signal `SIGCHLD` and sets `exit_state` to `EXIT_ZOMBIE`.
- Marks the task as dead and invokes the scheduler to execute a context switch.

Since the task exit value must be read by the parent, the `task_struct` itself is not freed yet. Only when such read takes place the memory holding the process descriptor is effectively returned to the memory subsystem.

3.2 The Scheduler

3.2.1 Scheduling Theory

The scheduler is a central component of any operating system. Its main job is to determine what is the next task to put on the CPU and for how long it should run. These decisions can have dramatic influence on the overall OS performance, as shown in [46, 28, 48, 11].

Multitasking OSes

In a multitasking system like a modern desktop computer, a server cluster or even some embedded systems like the Raspberry Pi, the scheduler must give the illusion that all the tasks are executing concurrently. On a uni-core system this is achieved by periodically switching task, while on multi-core systems multiple flow of execution are effectively executed in parallel. However, even in such systems, the number of cores available is much smaller than the usual thread set.

Depending on how task switches can be executed, it is possible to classify scheduling algorithms as *preemptive* or *non-preemptive*. In the latter case, the scheduler enters in

action only in two cases: the task on the CPU terminates its work, or voluntarily asks to be scheduled. The former case, on the other hand, allows the scheduler to replace a thread running on a core at any time, usually when an interrupt occurs. Consumer-grade operating system like Linux, Windows and macOS use preemptive schedulers.

Context Switch

The act of changing the active task on a CPU core is called *context switch*. The kernel saves the CPU context of the task currently running on the core in memory, suspending it, and restores the state of the task scheduled to run [40]. The state usually comprehend the integer and floating point register files, as well as the registers configuring the MMU, TLB and so on. In Linux, the code handling the process switch is contained in `context_switch()`, which is called only when the main scheduler function, `schedule()` is invoked.

If the threads involved in the switch belong to different processes, the memory address space must be changed. The consequence of this change is that the content of the caches, as well as the content of the TLB, is no longer valid and needs to be flushed.

As a result, a context switch can be a very expensive operation that must be reduced to the minimum [13]. At the same time, if context switches are not executed frequently, the other tasks may starve, leading to performance degradation. In this regard, switching between threads of the same process incurs in a lower performance penalty, as they share the memory address space.

Priorities

Many operating system, Linux included, support some form of priority-based scheduling. The *priority* represents the importance of a task within the system, an information the scheduler uses to determine how often the thread should run and, ideally, for how much time.

Linux employs two different priority mechanism, one dedicated to *normal* processes and another one dedicated to *real-time* tasks. The latter is a particular class of programs where a computation must end before a *deadline* expires. If it is unable to complete its job in time, there may be consequences with various degrees of severity. Examples of real-time tasks range from the audio thread of a music player (a deadline miss results in an audio glitch) to the software controlling the altitude of a plane (in this case, missing the deadline may lead to catastrophic consequences).

Linux assigns to each standard task a `nice` value, which describes the *niceness* of this task with respect to the others. `nice` is an integer number in the range $-20 \leq nice \leq 19$. A value of -20 indicates that the task is unwilling to share CPU with the others, hence it corresponds to a high priority value. Conversely, 19 is the lowest possible priority. Users can modify the `nice` of a task only if is the owner, otherwise it must be root.

Real-time tasks, on the other hand, use a completely different convention: for them, $0 \leq priority \leq 99$, with 0 being the lowest priority and 99 the maximum.

Internally the kernel adopts yet another convention, using an integer in the range $0 \leq priority \leq 139$. Here, 0 corresponds to the maximum priority value, while 139 is the minimum. The first 100 levels are assigned to real time tasks, while the other 40, corresponding to the size of `nice`'s range, are dedicated to normal tasks.

Tasks Classification

Tasks, from the scheduler's point of view, are usually classified as *CPU-bound* or *I/O-bound*. The former label is attributed to threads making relatively few I/O calls, spending most of the time executing code. An example may be a video encoder or a scientific application. The latter, instead, is given to threads executing small CPU burst and pausing often to wait for some I/O to complete. Examples of this category are GUI application, or a thread accessing a large file on disk. Of course, most of the real programs exhibit both characteristics, alternating I/O phases to compute phases.

From the discussion above, it is clear that an I/O-bound task almost never consumes its timeslice entirely, *yielding* the CPU control after a short amount of time. Conversely, a CPU-bound thread almost never releases the CPU voluntarily, hence it will be preempted by the scheduler after a system clock tick.

A scheduling algorithm designed with fairness as the main goal, such as the Linux CFS, guarantees equity by scheduling often I/O-bound threads, which will leave the CPU quickly, and then giving the remaining CPU time to compute-intensive tasks.

Load Balancing

In a multi-core system, it is rather inefficient having a global list of runnable threads. In fact, being a shared resource, the list needs to be protected by locks to maintain a coherent state. Thus, only one core at the time would be allowed to operate on the list, forcing its siblings to wait doing nothing useful for the end user. This *contention* problem becomes more evident as the number of parallel execution units grows. Moreover, operating on shared resources triggers the *cache coherency protocols*, which invalidate the cache lines of all the cores who recently accessed the data in question. These protocols are expensive and lead to more cache misses, further penalizing the performance.

Therefore, Linux and many other kernels prefer to keep frequently accessed resources, like the list of runnable tasks, local to each core. This approach removes almost any locking overhead and improves cache accesses, as every core can independently work on its private data. Remaining in the same list is beneficial also for the tasks themselves, because it is likely that the data on which they are working is still in the core's private cache when they are rescheduled.

However, the outlined solution can result in work unfairness among the cores. Without any form of data sharing, a core may be overloaded while others are idle. Of course, this situation must be avoided at all costs, hence OSes periodically check the load of each

core and, if necessary, *migrate* tasks from a core to the other. This operation is called *load balancing*.

Load balancing algorithms exist in two flavors: *push migration* and *pull migration* [40]. In the former, a dedicated kernel task looks for imbalance in any core. If it exists, some tasks are taken from the overloaded core list and *pushed* to other idle or less busy cores. In the latter, it is the core itself, when it becomes idle, to *pull* a task from a busier sibling.

The Linux kernel expolits both approaches: at each system clock tick a special, high priority task is started to balance the workload across the system. At the same time, whenever the `idle` task is executed, it tries to steal tasks from another core.

3.2.2 Scheduling Classes

The Linux scheduler is implemented with a modular approach. There is a *core* component, defined in `<kernel/sched/core.c>`, which makes high-level decisions, and there are various *scheduling classes* implementing the actual scheduling policies. Because of this structure, it is relatively trivial to add new policies with minimal modifications to the existing infrastructure.

A scheduling class is a C structure defined in `<include/linux/sched/sched.h>`. Listing 3.1 reports a partial list of its fields.

```
1  struct sched_class {
2      const struct sched_class *next;
3
4      void (*enqueue_task) (struct rq *rq, struct task_struct *p, int
       flags);
5      void (*dequeue_task) (struct rq *rq, struct task_struct *p, int
       flags);
6      void (*yield_task)   (struct rq *rq);
7      ...
8      void (*check_preempt_curr)(struct rq *rq, struct task_struct *p,
       int flags);
9      struct task_struct * (*pick_next_task)(struct rq *rq,
10                                            struct task_struct *prev,
11                                            struct rq_flags *rf);
12     ...
13 #ifdef CONFIG_SMP
14     int (*balance)(struct rq *rq, struct task_struct *prev, struct
       rq_flags *rf);
15     ...
16 #endif
17
18     void (*task_tick)(struct rq *rq, struct task_struct *p, int queued
       );
19     ...
20     void (*update_curr)(struct rq *rq);
21     ...
```

```
22 };
```

Listing 3.1: Scheduling class structure

`sched_class` provides a generic interface for the *core* component to interface with scheduling classes, which implement their policies by linking the class' functions to the structure's fields. A short explanation of the fields is given below:

- `next`: it is a pointer to the next scheduling class. It allows the core component to iterate over the various scheduling classes respecting the class' priority, for example when the next task to run must be selected.
- `enqueue_task`: adds a task to the list of runnable tasks held by the runqueue (runqueues are detailed in section 3.2.3). It is usually called when a task changes its state in `TASK_RUNNING`.
- `dequeue_task`: removes a task from the runqueue, for example because its `state` has changed from `TASK_RUNNING` to `TASK_INTERRUPTIBLE` or it has ended its execution.
- `yield_task`: the task voluntarily yields the CPU.
- `check_preempt_curr`: checks if the task currently running must be preempted. If so, it triggers a scheduling event. This function is invoked during task migrations or a task wake-up.
- `pick_next_task`: returns the next, most eligible task to be scheduled on the CPU.
- `balance`: this field exists only for simultaneous multi-processing system. It runs a load balancing pass, if required, before `pick_next_call` is invoked, to reduce the chances of scheduling the `idle` task.
- `task_tick`: accounts various statistics of the task currently running on the CPU and checks if it must be preempted.
- `update_curr`: similar to `task_tick`, however it is invoked at a much higher frequency.

3.2.3 Runqueues

A *runqueue* is a per-CPU data structure where the kernel stores the list of runnable tasks and other information related to scheduling and accounting purposes. Listing 3.2 presents some of its relevant fields, with the full definition available in `<include/-linux/sched/sched.h>`.

```
struct rq {
    raw_spinlock_t      lock;
    unsigned int        nr_running;
    ...
    struct cfs_rq       cfs;
    struct rt_rq        rt;
    struct dl_rq        dl;
    ...
    struct task_struct  *curr;
    ...
    int          cpu;
};
```

Listing 3.2: Runqueue structure

`lock` is the main runqueue lock. Every time the runqueue data is accessed, the lock must be obtained to maintain a valid state. One of the main purposes of this lock is to protect the list of runnable tasks during load balancing passes and migrations. To avoid deadlocks, the kernel defines the following locking rule: any code that wants to lock multiple runqueues must obtain the locks in ascending runqueue order. The order of the runqueue is specified by the CPU to which it belongs to, indicated by the `cpu` field.

The main runqueue holds other runqueues, each specific of a different scheduling class. In the default kernel v5.4.35, these are `cfs`, `rt` and `dl`. In these runqueues scheduling classes effectively store their list of runnable tasks. The kernel does not impose any restriction on the internal runqueues structure, hence scheduling classes are free to use any data structure suit their needs. Having runqueues nested in the main runqueue is necessary to maintain the abstraction layer between the core component and the scheduling classes. In fact, the former operates only on `struct rq`, and delegates the latter to access their own structures.

Finally, `curr` is a pointer to the currently active task on the CPU. Having a fast way to get this data is vital to the kernel, as this information is required for accounting and scheduling purposes.

3.2.4 Timers

Classical scheduling algorithms revolve extensively around timers, and the Linux scheduler makes no exception. Their main goals are to periodically trigger the scheduler to preempt a task when its timeslice expires and to allow accounting opportunities.

The Linux scheduler sets up the system timer to fire interrupts with frequency `HZ`, a constant quantity defined at compilation time. The default value is set to 250, although values in the range $[100,\ 1024]$ are considered valid too [7]. Every time the timer triggers an interrupt, the function `scheduler_tick`, defined in `kernel/sched/core.c`, is invoked. The function is not worth to be reported here, but essentially it does some bookkeeping and statistics accounting, triggers the load balancing algorithm if needed and calls the `task_tick` function of the current task's scheduler class.

The scheduler, however, supports an optional feature that adds a second, more precise source of interrupts: *high resolution timers*, or `hrtimers` [16]. Every main runqueue possesses a `hrtimer` exploited by scheduling classes to receive accurate preemption ticks [49]. In this case, the core scheduler function responding to the interrupt is `hrtick`, which runs in an hardware interrupt request context with interrupts disabled. The body of the function is rather small, as shown in listing 3.3:

```
static enum hrtimer_restart hrtick(struct hrtimer *timer)
{
    struct rq *rq = container_of(timer, struct rq, hrtick_timer);
    struct rq_flags rf;

    WARN_ON_ONCE(cpu_of(rq) != smp_processor_id());

    rq_lock(rq, &rf);
    update_rq_clock(rq);
    rq->curr->sched_class->task_tick(rq, rq->curr, 1);
    rq_unlock(rq, &rf);

    return HRTIMER_NORESTART;
}
```

Listing 3.3: hrtick

The function locks the runqueue, updates the runqueue's clock and invokes the `task_tick` hook of the scheduling class handling the current task. Here lies an important difference with the same call executed by `scheduler_tick`: the last parameter of `task_tick()`, `queued`, is set to 1, while in the previous case is set to 0. `queued` allows scheduling classes making use of `hrtimers` to distinguish the context from which `task_tick()` has been invoked.

3.2.5 The schedule function

The main and only entry point to schedule a thread is the function `schedule()`, located in `<kernel/sched/core.c>`. Usually, this function is entered when a thread blocks and release the CPU voluntarily, for example because of a mutex or is waiting for some I/O, or when the timeslice of the currently running task expires.

Due to the large number of operations, the subsequent listings do not report the whole functions, only the salient parts. The core of `schedule()` is reported in listing 3.4:

```
do {
    preempt_disable();
    __schedule(false);
    sched_preempt_enable_no_resched();
} while (need_resched());
```

Listing 3.4: schedule

Before entering the actual scheduling code the kernel disables preemption, because it has to interact with many per-CPU variables not protected by any lock (after all, every CPU works only on its copy). Preemption is enabled again once the scheduling process concludes its operation.

Commonly, when preemption is enabled again, the kernel invokes the scheduler to check if it should have been preempted in the meantime. However, since in this case preemption was disabled specifically to schedule a new task, preemption is enabled without the check.

The code executes in a loop to catch possible preemption opportunities missed while `__schedule` execution.

__schedule

`__schedule` is the main scheduler routine actually executing the scheduling logic. Listing 3.5 shows the function skeleton . It is important to outline that, for compactness, some function calls have been excluded from the listing.

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

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    local_irq_disable();
    ...
    rq_lock(rq, &rf);
    ...
    next = pick_next_task(rq, prev, &rf);
    clear_tsk_need_resched(prev);
    clear_preempt_need_resched();

    if (likely(prev != next)) {
        ...
        /* Also unlocks the rq: */
        rq = context_switch(rq, prev, next, &rf);
    } else {
        ...
        rq_unlock_irq(rq, &rf);
    }

    balance_callback(rq);
```

```
}
```

Listing 3.5: __schedule

The function immediately identifies the CPU on which is running to retrieve the related main runqueue. Then, to avoid interruptions in the critical path, it disables the local core interrupts. Furthermore, it obtains the main runqueue lock to prevent race conditions with the load balancing and task migrations algorithms. The scheduler then selects the next task to run through `pick_next_task()`, described in more details later on. It also clears scheduling flags to not loop in `schedule()`. Eventually, the kernel checks if the task being scheduled out is the same being scheduled in: if the test results false it executes a context switch, otherwise it unlocks the main runqueue and returns to `schedule()`, as the same task is still eligible for running.

Choice of the Next Task

`pick_next_task()` is the function in charge of selecting the task with the highest priority to run. Thanks to the modularity granted by scheduling classes, the resulting code (reported in listing 3.6) is elegant and compact.

```
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)) {

        p = fair_sched_class.pick_next_task(rq, prev, rf);
        if (unlikely(p == RETRY_TASK))
            goto restart;

        /* Assumes fair_sched_class->next == idle_sched_class */
        if (unlikely(!p))
            p = idle_sched_class.pick_next_task(rq, prev, rf);

        return p;
    }

restart:
#ifdef CONFIG_SMP
    for_class_range(class, prev->sched_class, &idle_sched_class) {
        if (class->balance(rq, prev, rf))
            break;
    }
#endif
```

```
29
30      ...
31
32      for_each_class(class) {
33          p = class->pick_next_task(rq, NULL, NULL);
34          if (p)
35              return p;
36      }
37
38      /* The idle class should always have a runnable task: */
39      BUG();
40  }
```

Listing 3.6: pick_next_task

The kernel first tries to reduce the execution time by exploiting a common case scenario: if every task in the main runqueue belongs to either the CFS class (the standard class for normal tasks) or the idle class, it ignores the other classes and it directly invokes the CFS's `pick_next_task()`. If it fails, it falls back to the idle class as it is guaranteed to possess a task to schedule.

On the other hand, if the tasks belong to different classes, the code gives the opportunity to each scheduling class to run a balancing pass. Subsequently, it iterates over every scheduling class following their priority and calling the related pick function. As soon as a schedulable task is found, its pointer is returned to the caller.

The scheduler code makes wide usage of macros to improve loops' readability. This is the case, for example, of the for loops declared at line 24 and 32, which expand to:

```
1  #define sched_class_highest (&stop_sched_class)
2
3  #define for_class_range(class, _from, _to) \
4  for (class = (_from); class != (_to); class = class->next)
5
6  #define for_each_class(class) \
7      for_class_range(class, sched_class_highest, NULL)
```

These definitions can be found in `<include/linux/sched/sched.h>`.

3.3 Completely Fair Scheduler Implementation

The Linux Completely Fair scheduler, also known as CFS, is the default scheduler for normal processes, referred to as `SCHED_NORMAL` (Linux) or `SCHED_OTHER` (POSIX), since version 2.6.23 [29]. This section details the scheduler model and its implementation. Unless specified otherwise, the listings in this chapter show code taken from `<kernel/sched/fair.c>`.

3.3.1 The Completely Fair Model

The Completely Fair scheduler models an ideal, precise multitasking CPU [30, 29]. An ideal multitasking CPU is capable of executing n tasks concurrently, giving to each task $\frac{1}{n}$ of CPU power. For example, if $n = 2$, the CPU would run each task at 50% of power, that is, they run concurrently. A real processor, however, is capable of executing only one task at the time. Hence, CFS introduces the concept of *virtual runtime*, which represents how much time the task has run on the ideal multitasking processor. Practically, it is the actual execution time normalized by the number of runnable tasks. The scheduler uses it to determine the next task to put on the core. The target is to maintain the virtual runtime of every task close to each other (ideally, to the same value), therefore the scheduler always picks the task with the smallest runtime.

A potential solution to faithfully emulate the ideal multitasking processor on real hardware is to interleave tasks on the CPU for an infinitesimal amount of time. However, this is unfeasible because context switches have a non-negligible overhead. CFS solves this problem by assigning to each process a timeslice and, when it expires, schedules the task with the smallest virtual runtime. The timeslice computation differs from the one used in the previous schedulers, as it does not merely assign a value based on the priority, but calculates it as a function of the number of runnable threads in the system [30, 40].

More in details, the kernel defines a variable called *targeted latency*, that is the period in which every task should be scheduled once. `nice` determines the *weight* of the task, that is then divided by the weight of every runnable task. The result of the division represents the proportion of the targeted latency in which the thread is allowed to run.

From this definition it is clear that, with the number of runnable tasks tending to infinity, the timeslice of the tasks tends to 0, going back to the original problem. To ensure that this doesn't happen, the scheduler imposes a lower bound on the minimum time a task is allowed to run called *minimum granularity*. When CFS reaches this point it stops being completely fair, but is an intended behavior to mitigate the overhead of context switch. The default values for the targeted latency and the minimum granularity are $6\ ms * (1 + log_2(ncpus))$ and $0.75\ ms * (1 + log_2(ncpus))$ respectively. $1 + log_2(ncpus)$ is a factor extending these values in a multi-core environment. The rationale, as documented in the kernel code, is that the effective latency as perceived by the user decreases (in a non-linear way) with the number of cores available.

3.3.2 Scheduler Entities

CFS does not operate directly on tasks, but on *scheduler entities*. An entity may represent a task or a group of tasks, to which CFS operations are applied hierarchically [43]. The goal of this feature is to introduce group-awareness to the scheduler, ensuring that time is shared equally not only among tasks, but also among groups. As an example, assume a system with two users where are running 20 tasks, 1 belonging to the first user and the others 19 to the second one. If CFS only targeted task fairness, each thread

would get 5% of CPU time. While equity exists from the scheduler point of view, it does not from the user side. With the introduction of scheduler entities, CFS first splits the CPU time between the two users, then goes down the hierarchy and further divides the CPU time among the tasks. Hence, each user would receive 50% of CPU time. The single task of the first user receives the entire 50%, while the tasks of the second one have to divide the 50% between themselves, therefore every thread would get around 2.6% of CPU time. The kernel does not place any bound on the hierarchy size, so it is possible to create complex structures as needed.

Within the kernel code, scheduler entities are implemented in `sched_entity`, a C structure declared in `<include/linux/sched.h>`. Listing 3.7 resports some of its fields.

```
struct sched_entity {
    /* For load-balancing: */
    struct load_weight      load;
    ...
    struct rb_node          run_node;
    struct list_head        group_node;
    unsigned int            on_rq;

    u64             exec_start;
    u64             sum_exec_runtime;
    u64             vruntime;
    ...
    struct sched_statistics     statistics;

#ifdef CONFIG_FAIR_GROUP_SCHED
    int             depth;
    struct sched_entity     *parent;
    /* rq on which this entity is (to be) queued: */
    struct cfs_rq           *cfs_rq;
    /* rq "owned" by this entity/group: */
    struct cfs_rq           *my_q;
#endif
    ...
};
```

Listing 3.7: Scheduler entity structure

Some of the fields are explained below:

- `run_node` is a pointer to the red-black tree node containing the entity. For more information about the tree and its usage in CFS, refer to section 3.3.3.
- `on_rq` is a flag specifying whether the entity is in a runqueue or not.
- `exec_start` stores the time instant in which the entity has been scheduled on the CPU.
- `sum_exec_runtime` accumulates the entity's execution time.

- `vruntime` represents the virtual runtime on the ideal multitasking CPU.
- `statistics` holds various accounting information.

`parent`, `cfs_rq` and `my_rq` struct members are employed when the group scheduling feature is enabled at compile time. As already stated, group scheduling in CFS is achieved by applying CFS operations recursively. Because of this, a scheduling entities may possesses its own runqueue, `my_q`, while contemporarily being contained in another runqueue, `cfs_rq`, owned by another entity, `parent`. With this organization it is trivial to recognize top-level and leaf entities (tasks): for the former `parent` is set to `NULL`, whilst for the latter `my_q` is set to `NULL`. `cfs_rq`, on the other hand, cannot be `NULL` in any case: a scheduling entity always belongs to a runqueue, be it the main one or the one owned by another entity.

Load calculation

`load` corresponds to the *weight* of the entity. The weight is derived from the `nice` value and serves as the actual priority in CFS. Its structure is rather simple, as shown in listing 3.8. It contains two fields, `weight` and `inv_weight`. The latter is the inverse of the former and is cached for performance reasons, as divisions typically require more clock cycles to compute than multiplications.

```
1 struct load_weight {
2     unsigned long   weight;
3     u32             inv_weight;
4 };
```

Listing 3.8: load_weight structure

The conversion `nice-weight`, executed in `set_load_weight()` in the core component, is rather simple: the kernel calculates the priority as:

```
int prio = p->static_prio - MAX_RT_PRIO;
```

where `p` is a pointer to a `task_struct`, `static_prio` is the priority statically assigned by the user and `MAX_RT_PRIO` is a constant representing the maximum priority value for a real time task, hence its value is 100. Essentially, it scales `nice` from the range $[-20,\ 19]$ to $[0,\ 39]$. `prio` is an index for accessing two look-up tables, `sched_prio_to_weight` and `sched_prio_to_wmult`, as:

```
load->weight = scale_load(sched_prio_to_weight[prio]);
load->inv_weight = sched_prio_to_wmult[prio];
```

The look-up tables provide a direct mapping between the scaled `value` and the weights. Their content is listed in 3.9 and 3.10. The values are stored in a fixed-point format where the 10 least significant bits represent the fractional part, therefore the unit value corresponds to 1024. An increase or decrease of `nice` level increases or reduces the CPU usage by 10%. The 10% change is relative, meaning that if there are two tasks

with `nice` level 0 and 1 or `nice` level 14 and 15 the difference in CPU usage is always 10%.

`scale_load` is a macro which, only on 64 bits machines, shifts `weight` by additional 10 bits to increase the resolution of the fractional part from 10 to 20 bits.

```
const int sched_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,
};
```

Listing 3.9: sched_prio_to_weight

```
const u32 sched_prio_to_wmult[40] = {
    /* -20 */     48388,     59856,     76040,     92818,    118348,
    /* -15 */    147320,    184698,    229616,    287308,    360437,
    /* -10 */    449829,    563644,    704093,    875809,   1099582,
    /*  -5 */   1376151,   1717300,   2157191,   2708050,   3363326,
    /*   0 */   4194304,   5237765,   6557202,   8165337,  10153587,
    /*   5 */  12820798,  15790321,  19976592,  24970740,  31350126,
    /*  10 */  39045157,  49367440,  61356676,  76695844,  95443717,
    /*  15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};
```

Listing 3.10: sched_prio_to_wmult

3.3.3 CFS runqueue

CFS selects as the next task to run the one with the smallest `vruntime`. The rationale is that the task with the smallest `vruntime` had less opportunities to run, creating an imbalance with respect to the ideal model. To achieve this, the scheduler does not use the classical concept of runqueue as a list of processes, but stores the runnable tasks in a red-black tree sorted by `vruntime` representing the execution timeline [34].

Red-Black Trees

A red-black tree is a balanced binary search tree supporting search, insertion and deletion operations in $\mathcal{O}(\log n)$ time. An example of the data structure is shown in figure 3.4.

A binary search tree is a red-black tree if the following properties are satisfied [2]:

1. each node has either a red or black color.

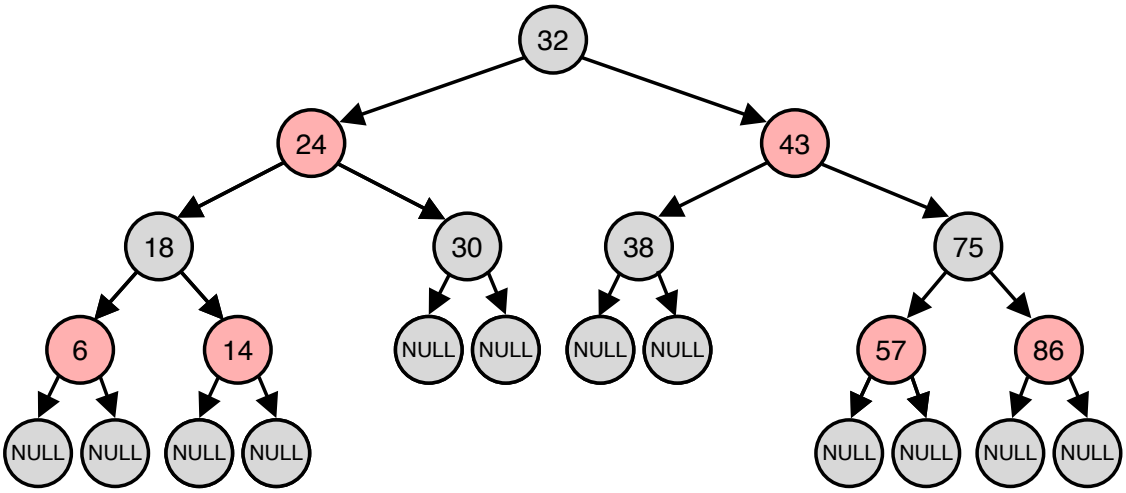


Figure 3.4: Red-black tree example

2. the root of the tree is black.
3. every leaf is black and its content is NULL.
4. the children of a red node are blacks.
5. every path from a node to its leaves contains the same number of black nodes.

From these definitions it is possible to demonstrate that a tree with n nodes has height $\leq 2log_2(n+1)$, hence the upper bound of $\mathcal{O}(\log n)$ for the search operation.

The kernel's tree declaration can be found in `<include/linux/rbtree.h>` and its implementation in `<lib/rbtree.c>`. There are two versions of the red-black tree: cached and non-cached. The difference consists in an additional pointer added to the root structure that maintains a reference to the leftmost node, that as shown in figure 3.4 corresponds to the scheduling entity having the smallest vruntime. The pointer grants a $\mathcal{O}(1)$ access time instead of the standard $\mathcal{O}(\log n)$, a useful speed-up in a critical path like the one present in `schedule()`.

CFS runqueue structure

The CFS runqueue structure is defined in `<include/linux/sched/sched.h>`. A partial list of the fields is given in listing 3.11.

```
1 struct cfs_rq {
2     struct load_weight  load;
3     unsigned long       runnable_weight;
4     unsigned int        nr_running;
5     ...
6     u64         min_vruntime;
```

```
7       ...
8       struct rb_root_cached   tasks_timeline;
9
10      struct sched_entity *curr;
11      ...
12      struct rq       *rq;
13      ...
14  };
```

Listing 3.11: CFS runqueue structure

Here is given an explanation of the fields:

- `nr_running` stores the number of runnable scheduling entities in the runqueue.
- `tasks_timeline` is the cached version of the red-black tree defining the execution timeline of the scheduling entities.
- `curr`, a pointer to the entity currently running on the CPU. If no entity of the runqueue is in execution, this field is set to `NULL`.
- `rq` is the pointer to the general, per-CPU runqueue.

As for scheduling entities, also CFS runqueues have a `load`. However, unlike entities, the load stored in a runqueue is cumulative. Whenever an entity is added or removed from the runqueue, its load is added or removed as well from the queue. This parameter is of great importance for some CFS operations explained later on.

`min_vruntime` is a monotonically increasing variable storing the minimum `vruntime` of all scheduling entities present in the runqueue. It normalizes the virtual runtime of entities joining or exiting the runqueue. This step is fundamental to guarantee both correct operations and fairness in CFS. Every entity logs in `vruntime` its execution time. When it exits a runqueue, the virtual runtime is not updated anymore. Once the entity re-enters a runqueue, its `vruntime` will be much lower than the one of the other entities, therefore the entity would receive an unfair amount of CPU time to match its siblings, leading to unfairness. The solution adopted by the kernel is to remove `min_vruntime` from the scheduling entity's `vruntime` whenever it exits the runqueue, and to add it again when it joins it.

3.3.4 Entity Update

CFS tracks `vruntime` to emulate the execution time spent on the ideal multitasking processor. It represents how much the task has run and, for the thread possessing the CPU, how much time it can still run. It is mandatory, then, to maintain an accurate value to faithfully reproduce the ideal system on real hardware. For this reason, CFS updates the runtime statistics of the currently running entity as many times as possible with `update_curr()`. The function is called, for example, by the handler of the system

timer when a tick occurs, or when a new entity is being enqueued or dequeued in the runqueue, or again after serving an interrupt. Its definition is shown in listing 3.12.

```
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if (unlikely(!curr))
        return;

    delta_exec = now - curr->exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;

    curr->exec_start = now;

    schedstat_set(curr->statistics.exec_max,
                  max(delta_exec, curr->statistics.exec_max));

    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq->exec_clock, delta_exec);
    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq);

    if (entity_is_task(curr)) {
        struct task_struct *curtask = task_of(curr);

        trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
        cgroup_account_cputime(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec);
    }

    account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

Listing 3.12: update_curr

The code stores in `delta_exec` the real execution time in nanoseconds. After some accounting, it increments the current entity's `vruntime` with the return value of `calc_delta_fair()`, reported in listing 3.13.

```
static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
{
    if (unlikely(se->load.weight != NICE_0_LOAD))
        delta = __calc_delta(delta, NICE_0_LOAD, &se->load);

    return delta;
}
```

Listing 3.13: calc_delta_fair

Its job is rather easy: if the entity has a weight equal to the `nice` level 0 nothing is done and `delta_exec` is directly added to `vruntime`. On the other hand, if the entity has a `nice` value other than 0, the function invokes `__calc_delta()` to compute the actual delta to be added to `vruntime`. Listing 3.14 reports the definition on `__calc_delta()`.

```
static u64 __calc_delta(u64 delta_exec, unsigned long weight, struct
    load_weight *lw)
{
    u64 fact = scale_load_down(weight);
    int shift = WMULT_SHIFT;

    __update_inv_weight(lw);

    if (unlikely(fact >> 32)) {
        while (fact >> 32) {
            fact >>= 1;
            shift--;
        }
    }

    /* hint to use a 32x32->64 mul */
    fact = (u64)(u32)fact * lw->inv_weight;

    while (fact >> 32) {
        fact >>= 1;
        shift--;
    }

    return mul_u64_u32_shr(delta_exec, fact, shift);
}
```

Listing 3.14: __calc_delta

`__calc_delta()` updates the `inv_weight` field of the `load_weight` structure pointed by `lw`. Since CFS calculations are executed in fixed point, it adjusts `fact` to have a value fitting 32 bits, and accounts the possible adjustments in `shift`. Then, it multiplies `fact` with the inverse of the load, which is equivalent to:

$$fact = \frac{fact}{lw \rightarrow weight} \tag{3.1}$$

Since a multiplication between two 32 bits variable requires up to 64 bits to be stored, `fact` may need to be scaled again and `shift` is updated accordingly. Finally, the code calls `mul_u64_u32_shr()`, a function implementing a fixed-point version of the multiplication, that is `(delta_exec * fact) >> shift`.

By putting everything together, without considering the bit shifting, the calculation executed by `__calc_delta()` essentially boils down to:

$$delta = \Delta_{exec} \cdot \frac{weight}{lw \rightarrow weight} \tag{3.2}$$

This means that the `delta` returned by `calc_delta_fair()` is the actual execution time weighted by the priority of the entity. In fact, by expanding everything in `update_curr()`, the `vruntime` update looks like:

$$vruntime(t) = vruntime(t-1) + \Delta_{exec} \cdot \frac{NICE\_0\_LOAD}{curr \rightarrow load} \tag{3.3}$$

Where `NICE_0_LOAD` is equal to the weight of `nice` level 0 and `curr` is the pointer to the current scheduling entity.

`vruntime` progresses with a different rate for tasks having a different `nice` value. The virtual runtime of a high priority thread increases more slowly than the one using a standard priority, while the opposite is true for a low priority task. The consequence is that the former has more chances to be rescheduled soon, while the latter ends up more easily in the right side of the CFS entity tree.

Once `vruntime` is updated the function calls `update_min_vruntime()`, which updates the runqueue's `min_vruntime` field ensuring that it never goes backward in time.

3.3.5 CFS Timeslice

CFS computes the timeslice as a proportion of the CPU time between the weights of the current thread and its runqueue. The kernel function in charge of this operation is `sched_slice()`, reported in listing 3.15.

```
1  static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
2  {
3  u64 slice = __sched_period(cfs_rq->nr_running + !se->on_rq);
4
5  for_each_sched_entity(se) {
6      struct load_weight *load;
7      struct load_weight lw;
8
9      cfs_rq = cfs_rq_of(se);
10     load = &cfs_rq->load;
11
12     if (unlikely(!se->on_rq)) {
13         lw = cfs_rq->load;
14
15         update_load_add(&lw, se->load.weight);
16         load = &lw;
17     }
18     slice = __calc_delta(slice, se->load.weight, load);
19 }
```

```
20 return slice;
```

Listing 3.15: sched_slice

`sched_slice()` takes as input a scheduling entity and the runqueue it belongs to. It stores in `slice` the *targeted latency*, the output of `__sched_period()` whose definition is shown in listing 3.16.

```
1 static u64 __sched_period(unsigned long nr_running)
2 {
3     if (unlikely(nr_running > sched_nr_latency))
4         return nr_running * sysctl_sched_min_granularity;
5     else
6         return sysctl_sched_latency;
7 }
```

Listing 3.16: __sched_period

The targeted latency depends on the condition `nr_running > sched_nr_latency`: if the result of the test is negative, the latency assumes the default value, otherwise it is computed as the multiplication between the number of runnable tasks and the minimum granularity. In the latter case, the kernel is stretching the target latency to reduce the overhead of context switches when many tasks are schedulable at the cost of fairness. `sched_nr_latency` is kept at $\frac{targeted\ latency}{minimum\ granularity}$, equal to 8 with the stock parameters.

Going back to `sched_slice()`, the function walks the entity hierarchy to compute the timeslice proportion. In each iteration obtains the runqueue of the entity being considered. Then, in case the entity is not scheduled on the runqueue, it adds the entity's load to the runqueue's one for the purpose of the calculation. Finally, it updates `slice` with a call to `__calc_delta()`, already presented in listing 3.14. Once it reaches the top scheduling entity, that is, `se->parent == NULL`, the loop ends and the function returns the timeslice to the caller. Mathematically, the steps executed in the body of the for loop can be written as:

$$slice_i = slice_{i-1} \cdot \frac{se_i \rightarrow weight}{rq_{i+1} \rightarrow weight} \tag{3.4}$$

That is, a recursive equation where i is the entity index in the hierarchy, se_i a pointer to the i-th scheduling entity and rq_{i+1} the pointer to the runqueue owning se_i. Equation 3.4 can be expanded by substituting $slice_i$ in $slice_{i+1}$ for every scheduling entity. Let $slice_{-1}$ be the return value of `__sched_period()`, the equation can be rewritten as:

$$timeslice = slice_n = slice_{-1} \prod_{i=0}^{n} \frac{se_i \rightarrow weight}{rq_{i+1} \rightarrow weight} \tag{3.5}$$

Where n is the height of the entity hierarchy.

Timeslice computation example

To demonstrate how `sched_slice()` divides the scheduling period among the entities, figure 3.5 shows a simple system where the main runqueue owns two scheduling entities, and one of them possess a runqueue where other two entities are scheduled. The circles correspond to entities, the rectangles to runqueues.

(a) Entities with equal weights (b) Entities with different priorities

Figure 3.5: Timeslice example

In figure 3.5a every entity has the same `nice` level, 0, which corresponds to a `weight` of 1024. Assume that the input of the function are E2 and its runqueue R1 and let the default timeslice be $10\ ms$. By applying equation 3.4 in the first iteration, the result is:

$$slice_0 = slice_{-1} \cdot \frac{E2 \rightarrow weight}{RQ1 \rightarrow weight} = 10\ ms \cdot \frac{1024}{2048} = 5\ ms$$

In the second and last iteration, $slice_0$ is further divided as:

$$slice_1 = slice_0 \cdot \frac{E0 \rightarrow weight}{RQ0 \rightarrow weight} = 5\ ms \cdot \frac{1024}{2048} = 2.5\ ms$$

As expected, the timeslice assigned to E2 is $\frac{1}{4}$ of the scheduling period, because CFS divides equally the time between E0 and E1 and E0's timeslice is shared uniformly by E2 and E3.

In figure 3.5b the situation is different: E0 has a `nice` level equal to -5, which raises the entity's weight to 3021. In this case E0, and consequently E2 and E3, are entitled to a longer share of the CPU time. By executing the computation again, the timeslice assigned to E2 is:

$$slice_0 = slice_{-1} \cdot \frac{E2 \rightarrow weight}{RQ1 \rightarrow weight} = 10\ ms \cdot \frac{1024}{2048} = 5\ ms$$
$$slice_1 = slice_0 \cdot \frac{E0 \rightarrow weight}{RQ0 \rightarrow weight} = 5\ ms \cdot \frac{3021}{4045} \approx 3.73\ ms$$

Which is indeed higher than in the previous case. Of course, a timeslice increase for E2 (and consequently E3, since they have the same weight) must be compensated by a reduction of E1's timeslice to remain within the scheduling period's boundaries. In fact, if in the first example for E1:

$$slice_0 = slice_{-1} \cdot \frac{E1 \rightarrow weight}{RQ0 \rightarrow weight} = 10\ ms \cdot \frac{1024}{2048} = 5\ ms$$

In the second example, the slice is reduced to:

$$slice_0 = slice_{-1} \cdot \frac{E1 \rightarrow weight}{RQ0 \rightarrow weight} = 10\ ms \cdot \frac{1024}{4045} \approx 2.5\ ms$$

3.3.6 Scheduler Tick

The core scheduler invokes the scheduling class' `task_tick` whenever a timer interrupt occurs. The hook is mapped to `task_tick_fair()` for CFS, shown in listing 3.17.

```
static void task_tick_fair(struct rq *rq, struct task_struct *curr,
    int queued)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
    }

    if (static_branch_unlikely(&sched_numa_balancing))
        task_tick_numa(rq, curr);

    update_misfit_status(curr, rq);
    update_overutilized_status(task_rq(curr));
}
```

Listing 3.17: task_tick_fair

The function walks the scheduling entity, and for every entity calls `entity_tick()`:

```
static void
entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int
    queued)
{
    update_curr(cfs_rq);

    update_load_avg(cfs_rq, curr, UPDATE_TG);
    update_cfs_group(curr);

```

```
#ifdef CONFIG_SCHED_HRTICK
    if (queued) {
        resched_curr(rq_of(cfs_rq));
        return;
    }

    if (!sched_feat(DOUBLE_TICK) &&
            hrtimer_active(&rq_of(cfs_rq)->hrtick_timer))
        return;
#endif

    if (cfs_rq->nr_running > 1)
        check_preempt_tick(cfs_rq, curr);
}
```

Listing 3.18: entity_tick

As usual in CFS, the function updates the current entity's statistics with `update_curr()`, then updates the average load and the CFS group's weight associated to the entity. If the kernel has been compiled with support to `hrtick` timers, `queued` is evaluated to determine the origin of the tick: the runqueue's `hrtimer` or the system timer. In the former case, `entity_tick()` raises immediately a rescheduling event, because the `hrtimer` fires an interrupt only when the entity's timeline as calculated by `sched_slice()` (explained in section 3.3.5) expires. Otherwise, if the interrupt was triggered by the system timer, the kernel distinguishes two cases:

1. `CONFIG_SCHED_HRTICK` is set and the `DOUBLE_TICK` feature is not supported: the function ends its execution, since CFS is exploiting the `hrtimer`

2. `CONFIG_SCHED_HRTICK` is set and the `DOUBLE_TICK` feature is supported, or `CONFIG_SCHED_HRTICK` is not set: if the runqueue contains only the current task nothing is done, otherwise the scheduler checks for preemption opportunities.

Without `hrtimer` support, the scheduler must manually check at each tick if is time to preempt the current task using `check_preempt_update()`, defined in listing 3.19.

```
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    unsigned long ideal_runtime, delta_exec;
    struct sched_entity *se;
    s64 delta;

    ideal_runtime = sched_slice(cfs_rq, curr);
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    if (delta_exec > ideal_runtime) {
        resched_curr(rq_of(cfs_rq));
        clear_buddies(cfs_rq, curr);
        return;
```

```
14      }
15
16      if (delta_exec < sysctl_sched_min_granularity)
17          return;
18
19      se = __pick_first_entity(cfs_rq);
20      delta = curr->vruntime - se->vruntime;
21
22      if (delta < 0)
23          return;
24
25      if (delta > ideal_runtime)
26          resched_curr(rq_of(cfs_rq));
27  }
```

Listing 3.19: check_preemt_tick

The timeslice assigned to the entity is retrieved from `sched_slice()`, and the real execution time is stored in `delta`. The function then checks if the timeslice has expired and raises a scheduling request in case the test results true. The function uses the real execution time, instead of `vruntime`, because priority is already accounted in the timeslice calculation. To mitigate context switches' overhead, `check_preempt_tick()` terminates its execution if the amount of time spent on the CPU by the current entity is less than the minimum granularity. When also this test results false, the scheduler checks if the leftmost task deserves the CPU more than the current entity by comparing their `vruntimes`: if $delta < 0$ or $delta \leq ideal\ runtime$ the entity currently running on the CPU is still entitled to own it, otherwise the function raises the rescheduling flag.

3.3.7 Entity Enqueue

Whenever a task becomes active again, for example because a condition is verified, the core scheduler `enqueue_task()` is invoked. As for most of the operations, the core scheduler resorts to scheduling classes' functions to do the actual work. For CFS, the scheduling class hook `enqueue_task()` is mapped to `enqueue_task_fair()`. The function does a lot of accounting not relevant for the work of this thesis, therefore listing 3.20 reports only the interesting parts.

```
1  static void
2  enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags)
3  {
4      struct cfs_rq *cfs_rq;
5      struct sched_entity *se = &p->se;
6      int idle_h_nr_running = task_has_idle_policy(p);
7
8      ...
9
10     for_each_sched_entity(se) {
```

```
11          if (se->on_rq)
12              break;
13          cfs_rq = cfs_rq_of(se);
14          enqueue_entity(cfs_rq, se, flags);
15          if (cfs_rq_throttled(cfs_rq))
16              break;
17          cfs_rq->h_nr_running++;
18          cfs_rq->idle_h_nr_running += idle_h_nr_running;
19
20          flags = ENQUEUE_WAKEUP;
21      }
22
23      ...
24  }
```

Listing 3.20: enqueue_task_fair

The loop iterates over the entities hierarchy to enqueue entities in their associated runqueue with `enqueue_entity()`. It stops its execution when the top-level entity is added to the main per-core CFS runqueue, or if the scheduling entity is already to runqueue, or when the runqueue on which the scheduling entity has to be scheduled is throttled. For the purpose of this discussion, runqueue throttling is an optional CFS grouping extension enabled by `CONFIG_CFS_BANDWIDTH`. Entities groups can run only for a specified amount of time, after which are not scheduled for the rest of scheduling period, even if the CPU is idle. For more information, refer to [42].

The interesting work is performed by `enqueue_entity()`, presented in listing 3.21:

```
1  static void
2  enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int
       flags)
3  {
4      bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags &
       ENQUEUE_MIGRATED);
5      bool curr = cfs_rq->curr == se;
6
7      if (renorm && curr)
8          se->vruntime += cfs_rq->min_vruntime;
9
10     update_curr(cfs_rq);
11     if (renorm && !curr)
12         se->vruntime += cfs_rq->min_vruntime;
13
14     update_load_avg(cfs_rq, se, UPDATE_TG | DO_ATTACH);
15     update_cfs_group(se);
16     enqueue_runnable_load_avg(cfs_rq, se);
17     account_entity_enqueue(cfs_rq, se);
18
19     if (flags & ENQUEUE_WAKEUP)
20         place_entity(cfs_rq, se, 0);
21
```

```
22      check_schedstat_required();
23      update_stats_enqueue(cfs_rq, se, flags);
24      check_spread(cfs_rq, se);
25      if (!curr)
26          __enqueue_entity(cfs_rq, se);
27      se->on_rq = 1;
28
29      ...
30  }
```

Listing 3.21: enqueue_entity

At the beginning of the function, two boolean variables are declared, `renorm` and `curr`. The former indicates the need for a normalization of the entity's `vruntime`, the latter is set to true if the entity being enqueued is the one currently scheduled on the CPU. `curr` determines if the normalization, when required, must be performed before or after the call to `update_curr()`. If the task being enqueued is running on the CPU, it is mandatory to normalize its value *before* `update_curr()` to maintain a correct value in the runqueue's `min_vruntime`, as explained in section 3.3.4. Then, the weight of the entity is reflected in its CFS group and runqueue with calls to `update_cfs_group()` and `account_entity_enqueue()`, respectively. Eventually, after some accounting, if the task is not the one currently running it is added to the red-black tree by `__enqueue_entity()`.

3.3.8 Entity Dequeue

During their lifetime tasks may need to temporarily stop their execution, for example while waiting for I/O operations to occur or for a mutex to unlock. In these cases, the core scheduler function `dequeue_task()` is invoked. Eventually, the core scheduler resorts to the scheduling class' `dequeue_task()` hook which, for CFS, corresponds to `dequeue_task_fair()`. For the same reasons of the enqueue process exposed in section 3.3.7, `dequeue_task_fair()` mostly comprises account operations not relevant for this explanation, therefore listing 3.22 presents only the main part of the function.

```
1  static void dequeue_task_fair(struct rq *rq, struct task_struct *p,
       int flags)
2  {
3      struct cfs_rq *cfs_rq;
4      struct sched_entity *se = &p->se;
5      int task_sleep = flags & DEQUEUE_SLEEP;
6      int idle_h_nr_running = task_has_idle_policy(p);
7
8      for_each_sched_entity(se) {
9          cfs_rq = cfs_rq_of(se);
10         dequeue_entity(cfs_rq, se, flags);
11
12         if (cfs_rq_throttled(cfs_rq))
13             break;
```

```
14          cfs_rq->h_nr_running--;
15          cfs_rq->idle_h_nr_running -= idle_h_nr_running;
16          if (cfs_rq->load.weight) {
17              se = parent_entity(se);
18              if (task_sleep && se && !throttled_hierarchy(cfs_rq))
19                  set_next_buddy(se);
20              break;
21          }
22          flags |= DEQUEUE_SLEEP;
23      }
24
25      ...
26  }
```

Listing 3.22: dequeue_task_fair

The code is almost specular to 3.20, however it presents a subtle difference: if the entity is scheduled in a runqueue with a weight greater than 0, the loop is stopped as well. In this case, in fact, the owner of the runqueue still possesses runnable entities, hence it cannot be dequeued.

`dequeue_entity()`, presented in listing 3.23, actually dequeues the entity from the runqueue in which is contained.

```
1  static void
2  dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int
       flags)
3  {
4      update_curr(cfs_rq);
5
6      update_load_avg(cfs_rq, se, UPDATE_TG);
7      dequeue_runnable_load_avg(cfs_rq, se);
8
9      update_stats_dequeue(cfs_rq, se, flags);
10
11     clear_buddies(cfs_rq, se);
12
13     if (se != cfs_rq->curr)
14         __dequeue_entity(cfs_rq, se);
15     se->on_rq = 0;
16     account_entity_dequeue(cfs_rq, se);
17
18     if (!(flags & DEQUEUE_SLEEP))
19         se->vruntime -= cfs_rq->min_vruntime;
20
21     return_cfs_rq_runtime(cfs_rq);
22
23     update_cfs_group(se);
24
25     if ((flags & (DEQUEUE_SAVE | DEQUEUE_MOVE)) != DEQUEUE_SAVE)
26         update_min_vruntime(cfs_rq);
```

```
}
```

Listing 3.23: dequeue_entity

The function immediately calls `update_curr()` to update the current task's statistics and the runqueue's `min_runtime`. Subsequently, it removes the entity's weight from the runqueue and its CFS group to exclude it from the average load computation. Then, if the entity is not the one currently running on the CPU, it is removed from the red-black tree by `__dequeue_entity()`. Finally, `dequeue_entity()` removes `min_vruntime` from the entity's `vruntime` for the reasons explained in section 3.3.3 and calls again `update_vruntime()`. This second invokation is necessary due to the possibility that the entity being removed is the one holding back `min_vruntime`.

3.3.9 Pick the Next Entity

The core scheduler, as already mentioned in section 3.2.5, calls the scheduler class' `pick_next_task()` to select the next task to run. In CFS, `pick_next_task()` is binded to `pick_next_task_fair()`. The function is quite complicated when `CONFIG_FAIR_GROUP_SCHED` is enabled due to the amount of operations required to manage groups. However, at its core, follows the steps presented in listing 3.24, the default implementation when CFS groups are disabled.

```
static struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct
    rq_flags *rf)
{
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;
    struct task_struct *p;
    int new_tasks;

    ...

    if (prev)
        put_prev_task(rq, prev);

    do {
        se = pick_next_entity(cfs_rq, NULL);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);

    p = task_of(se);

done: __maybe_unused;
#ifdef CONFIG_SMP
    list_move(&p->se.group_node, &rq->cfs_tasks);
#endif

```

```
27      if (hrtick_enabled(rq))
28          hrtick_start_fair(rq, p);
29
30      update_misfit_status(p, rq);
31
32      return p;
33
34      ...
35
36  }
```

Listing 3.24: pick_next_task_fair

`prev` is a pointer to the task being scheduled out. If it is not `NULL`, the code calls `put_prev_task()` to insert the task in the runqueue it belongs to. The loop determines the next task to run: in contrast to the usual `for_each_sched_entity()`, which walks up the entity hierarchy, this `do...while` *descends* it. At every iteration, it picks the entity deserving more to run with `pick_next_entity()`, and sets it as `cfs_rq`'s current entity with `set_next_entity()`. This function also detaches the entity from the red-black tree, because the entity is about to change state from "runnable" to "running". `cfs_rq` is then updated to point to the runqueue *owned* by the current scheduling entity. The loop ends when `cfs_rq` points to `NULL`, that is, it has been found an entity representing a task. If `hrtimers` are in use, the function additionally configures the timer to fire an interrupt when the entity's timeslice expires. Finally, the code returns to the core scheduler the pointer of the task to run.

Entity Selection

`pick_next_entity()` selects the entity deserving more CPU time. Ideally, the scheduler aims to run the leftmost entity in the red-black tree, obtained from `__pick_first_entity()`:

```
1  struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
2  {
3      struct rb_node *left = rb_first_cached(&cfs_rq->tasks_timeline);
4
5      if (!left)
6          return NULL;
7
8      return rb_entry(left, struct sched_entity, run_node);
9  }
```

Listing 3.25: __pick_first_entity

The implementation is strikingly simple: the tree caches the leftmost element to speed-up the pick operation, hence it is enough to access the `leftmost` variable stored in `tasks_timeline`, which is exactly what the macro `rb_first_cached()` does. The function then returns the pointer to the `sched_entity` corresponding to the red-black tree node through the `rb_entry()` macro.

`pick_next_entity()`, however, may not select the entity with the smallest `vruntime`, overriding the decision of `__pick_next_entity()`. If it is possible to schedule again the task currently running on the CPU without being too unfair it does so, as the thread may benefit from cache locality.

Chapter 4

Methodology

This chapter details the methodology adopted to study PMC-based scheduling. The first step toward the objective, discussed in section 4.1, is to identify the CPU resources to be monitored whose effectiveness depend on thread behavior and scheduling ordering.

The scheduler must become aware of performance counter existence, so that it can instructs PMCs to monitor the selected resources and access the events count. For this purpose, section 4.2 presents a lightweight PMCs library developed specifically for use inside the kernel.

Finally, section 4.4 explains the mathematical models and the scheduler modifications required to implement PMC-based scheduling in the CFS.

4.1 Resource Identification

Modern superscalar microprocessors employ many mechanisms to deliver high performance to the end user. As an example, figure 4.1 shows the internal architecture of an Intel Sandy Bridge processor.

The processor has a classic CISC structure [24]: there is front end, that fetches instructions from memory and decodes them in microarchitectural instructions (μops), and a backend to execute them. The front end is capable of fetching multiple instructions at each clock cycle, exploiting the branch prediction unit and the trace cache [37] to continuously feed the pipeline even in presence of branches. Sandy Bridge CPUs are capable of decoding 4 μops per cycle, which are forwarded to the allocation queue and then to the reorder buffer to enable out-of-order execution [20]. μops are dispatched through 6 different ports to various execution units (EUs), each designed to support a different set of instructions. Figure 4.1 also shows the memory subsystem, where there are two dedicated L1 caches, one for instructions and one for data, connected to a unified L2 cache. The L2 cache is also connected to the L3 cache, which is shared by every core in the package and forwards requests to the main memory. This design is able to provide a clock per instruction (CPI) lower than 1, but the CPI may increase due to

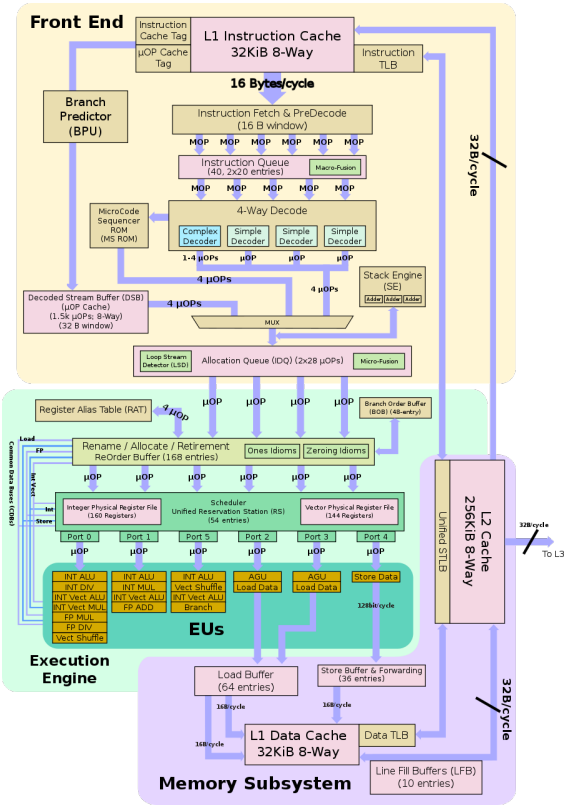


Figure 4.1: Intel Sandy Bridge Pipeline. Taken from [39].

pipeline stalls or flushes.

Caches have been integrated to mitigate the increasing performance gap between the CPU and main memory, and it is now not uncommon to have processors with a three-level cache hierarchy. Every cache level increases in capacity, but requires more clock cycles to fetch the data. Furthermore, is shared among more hardware threads, creating a contention problem.

Branch mispredictions are especially costly in deeply pipelined designs, since a flush results in many wasted clock cycles. Hardware designers have resorted to a wide range of techniques to improve predictors accuracy, such as saturating counters, branch history tables [47] and, more recently, machine-learning [25].

Although caches and branch predictors are able to improve performance, they have a common problem: the amount of information they can store is limited. The area budget dedicated to these resources is constrained by both performance and cost requirements. Cache and branch misses are inevitable for a single thread, and the situation becomes even worse in a multitasking environment, where these resources are shared by different threads. The usefulness of these hardware mechanisms is heavily influenced by how a task interacts with them. If a thread is particularly cache-intensive, the content previously loaded by other tasks may be flushed, reducing the probability of finding the

data again once they are rescheduled. Therefore, caches and branch predictors are valid candidates to study whether scheduling algorithms may improve the system throughput considering thread-hardware interactions.

4.1.1 PMCs selection

Cache

Caches are complex hardware structures. When a data miss occurs, the replacement algorithm is triggered to evict a cache line and store the new data. They are often paired with hardware prefetchers, whose job is to predict the next load address coming from the CPU for requesting the data before it is actually used, improving memory latency. In multi-core environments, caches also support coherency protocols to maintain a valid state when the same data is present in multiple caches [45, 27].

The cache performance monitoring unit provides insights on this information, such as the number of cache refills, write-backs, prefetcher requests, cache lines state, load hit or miss and so on. Due to the limited number of available counters, it is possible to monitor only a subset of events. The objective of this thesis is to enable PMC-based scheduling analyzing the tasks' execution profile, therefore only load events are considered. The list of events is available in table 2.2 of chapter 2.

Branch Predictor

Branches can be classified under many different categories: conditional or unconditional, near or far, calls or return, direct or indirect, just to make few examples. The branch predictor monitoring unit tracks and differentiates these events, allowing users to fine-tune performance analysis. In superscalar processor, another difference exists in speculative and retired events. To characterize the task interaction with the branch predictor, however, it is of interest to know the global number of retired predictions and mispredictions. For this reason, this work considers the architectural events `BR_INST_RETIRED.ALL_BRANCHES` and `BR_MISP_RETIRED.ALL_BRANCHES`.

4.2 PMCs library

The scheduler needs to access performance counters to monitor the tasks' execution profile and make PMC-based scheduling decisions. There already exist many PMCs libraries, but they are unsuitable for this work due to various reasons:

- *perf* [35]: the built-in Linux library for accessing performance counters from user-space. *perf*, due to its use-cases and structure, introduces too much overhead in a critical path like the scheduler code [44].

- *PAPI* [41]: a library supporting a diverse set of hardware and software platforms, providing a unified API for monitoring events. It is built for user-space operations over *perf* on Linux, therefore it adds even more overhead without offering kernel-space advantages.

- *PMCTrack* [38]: a loadable, architecture-agnostic kernel module, developed specifically for kernel development. Initially, the work of this thesis was based on this module, however it has two main drawbacks. The most important one is that, being a module, it is dynamically loaded only when some processes have already started, preventing the possibility to apply PMC-based scheduling at a global level. Moreover, its overhead cannot be neglected due to the amount of features supported not needed by this work, which impact both execution time and caches utilization.

This section describes a custom, lightweight PMCs library integrated in the core kernel, providing low execution and memory overheads API for accessing performance counters within the scheduler. It is designed to be easily extendible over different processors architectures and microarchitectures in a transparent way for the end user. The library works in two steps: first, every core must make a configuration call as soon as it boots to set up and start the performance counters. Then, it is possible to use the dedicated read function to obtain the number of events counted by a specific PMC.

4.2.1 Configuration

The number of performance counters and the events to monitor are defined at compilation time. The former is specified in a macro, `PMCS_USED`, that defines the size of a static array of `struct pmc_conf`. `pmc_conf` is a simple structure formed by two elements, the PMC index and the event such counter must monitor. To configure the monitoring units at runtime, it is enough to call the function `pmcs_init()` without specifying any parameter.

In a multi-core environment, logical cores have private copies of the PMCs set, hence `pmcs_init()` must be individually called by every execution unit. Moreover, this function must be invoked before the scheduler is initialized, otherwise the PMC values will always be 0, defeating the purpose of PMC-based scheduling.

Multi-cores system, when powered on, execute boot operations on a single core, known as the *boot core*. It is job of the software to eventually initialize and activate the other cores. Because of this, `pmcs_init()` is called from two different paths. For the boot core, the PMCs initialization happens inside `start_kernel()`, defined in `<init/main.c>`, just after the CPU ended its early boot operations. For the secondary cores, instead, the initialization takes place in `boot_secondary()`, defined in `<kernel/arch/x86/kernel/smpboot.c>` for the x86 architecture, that is the first function executed by a newly awakened core.

Internally, `pmcs_init()` calls an architecture-specific initialization function which, for Intel processors, corresponds to `init_intel_core_pmc()`. Listing 4.1 shows the function definition.

```
/* IA32_PERFEVTSEL register structure */
struct perfevtsel {
    volatile u64 ev_sel : 8;
    volatile u64 u_mask : 8;
    volatile u64 usr : 1;
    volatile u64 os : 1;
    volatile u64 edge : 1;
    volatile u64 pc : 1; /* reserved since Sandy Bridge arch */
    volatile u64 en_int : 1;
    volatile u64 reserved1 : 1;
    volatile u64 en : 1;
    volatile u64 inv : 1;
    volatile u64 cmask : 8;
    volatile u64 reserved2 : 32;
};

static __always_inline void init_intel_core_pmcs(void)
{
    struct perfevtsel perfevtsel;
    u64 *p;
    u64 v;
    int i;

    for (i = 0; i < PMCS_USED; i++) {
        switch (pmcs_conf[i].pmc_evt) {
        case UNHALTED_CORE_CYCLES_EVT:
            perfevtsel.ev_sel = EV_SEL_UNHALTED_CORE_CYCLES;
            perfevtsel.u_mask = UMASK_UNHALTED_CORE_CYCLES;
            break;

        ...
        }

        perfevtsel.usr = COUNT_USR_MODE_EN;
        perfevtsel.os = COUNT_OS_MODE_EN;
        perfevtsel.edge = EDGE_DETECT_DIS;
        perfevtsel.pc = PIN_CONTROL_DIS;
        perfevtsel.en_int = INT_DIS;
        perfevtsel.reserved1 = 0;
        perfevtsel.en = EN_COUNT;
        perfevtsel.inv = NO_INV_CMASK;
        perfevtsel.cmask = 0;
        perfevtsel.reserved2 = 0;

        p = (u64 *)&perfevtsel;
        v = *p;
        wrmsrl(BASE_PERFEVTSEL_ADDR + (int)pmcs_conf[i].pmc, v);
```

```
48      }
49  }
```

Listing 4.1: init_intel_core_pmc

`struct perfevtsel` is a C structure matching the layout of the IA32_PERFVTSELx MSRs, detailed in section 2.2.2. The code iterates over the static array of `struct pmc_conf` to set the *event select* and *unit mask* of `perfevtsel` accordingly. The rest of the structure initialization is common to every performance counter: events are counted both in user and kernel mode, the counter is immediately started and overflow interrupts are disabled. Eventually, the function converts the structure's bit field to a 64 bits unsigned number and updates the MSR specified by the configuration structure with `wrmsrl()`, the built-in kernel function to write MSRs. The resulting assembly, not reported here for compactness, is very efficient: `perfevtsel` is converted to a 64-bit value, of which the lower 16 bits, corresponding to the event select and the unit mask, are written by two `mov` instructions on 8-bit operands. The remaining structure initialization is compressed into a fixed hexadecimal value, `0x430000`, which is put in OR with the result of the switch and then written in the IA32_PERFVTSELx register.

4.2.2 Read Access

Once configured, performance counters can be accessed at any time with the `read_pmc()` function, which accepts the performance counter index as parameter. As for `pmcs_init()`, `read_pmc()` resorts to an architecture-specific function to actually read the register content, that is `read_intel_core_pmc()` for Intel architectures. Listing 4.2 shows the function definition.

```
1 static __always_inline u64 read_intel_core_pmc(pmc_t pmc)
2 {
3     u64 v;
4     rdmsrl(BASE_PMC_ADDR + (int)pmc, v);
5     return v;
6 }
```

Listing 4.2: read_intel_core_pmc

The function's body is very simple: `rdmsrl()` is a kernel macro storing in `v` the content of the MSR located at the specified address. Such a short function keeps the overhead at a bare minimum, which is extremely valuable since it is called in performance critical paths like context switches. It also has a minimal impact on caches, avoiding interference with measurements.

4.3 Fixed Point Format

Fixed point is a method for representing rational numbers. As the name suggests, a fixed amount of bits is assigned to the integer and fractional parts. In contrast with floating

point, which requires dedicated hardware like a FPU to provide acceptable performance, fixed point numbers can be processed on a standard integer ALU, that is faster and less power hungry. However, due to the limited number of bits dedicated to the fraction, precision may become a concern.

The fractional part is represented in the same base b as the integer one, but uses negative powers. Figure 4.2 shows the the bit layout of fixed point number on $n_b = 8$ bits using $p = 4$ least significant bits to store the fraction.

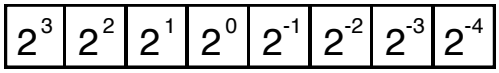


Figure 4.2: Fixed point power values

The fixed point format represents a number n as an integer value v implicitly scaled by a fixed factor b^{-p}, that is:

$$n = v \cdot b^{-p} = \frac{v}{b^p} \tag{4.1}$$

The factor may assume any value, however when using this format with binary number is convenient to use a power of 2 for performance reasons, because multiplications and divisions are optimized into shifts. For example, $n = 1.5$ can be stored in figure 4.2 as $v = n \cdot b^p = 1.5 \cdot 2^4 = 24$, which in binary translates to 0001 1000.

4.3.1 Fixed Point Arithmetics

Let $x = n_1 \cdot 2^{-p}$ and $y = n_2 \cdot 2^{-p}$ be two different fixed point numbers using the same base and numbers of bits to store the fraction. Additions and subtractions are then naturally defined as:

$$x + y = n_1 \cdot 2^{-p} + n_2 \cdot 2^{-p} = (n_1 + n_2) \cdot 2^{-p} \tag{4.2}$$
$$x - y = n_1 \cdot 2^{-p} - n_2 \cdot 2^{-p} = (n_1 - n_2) \cdot 2^{-p} \tag{4.3}$$

which, unless an overflow occurs, produces the exact mathematical result. For multiplications and divisions, however, there is need for additional care. Consider the multiplication between x and y:

$$x \cdot y = (n_1 \cdot 2^{-p}) \cdot (n_2 \cdot 2^{-p}) = (n_1 \cdot n_2) \cdot 2^{-2p} \tag{4.4}$$

The fractional part is no longer using p bits, but $2p$. Consequently, the number is not stored anymore in the expected format and needs to be corrected with adding an additional multiplication by 2^p:

$$(n_1 \cdot n_2) \cdot 2^{-2p} \cdot 2^p = (n_1 \cdot n_2) \cdot 2^{-p} \tag{4.5}$$

To show the correctness of the operation, let k = `0x0F80` and l = `0x02C0` two unsigned fixed point numbers on 16 bits using p = 8. In decimal, $k = 15.5$ and $l = 2.75$. The result of the multiplications between k and l is `0x2AA000`, a value which fits only in 32 bits (assuming to be on a standard consumer CPU architecture). Dividing `0x2AA000` by 2^{16} leads to 42.625, the expected result using the decimal notation. Hence, the multiplication output is effectively using 16 bits to store the fractional part. The correction factor 2^p can be implemented as a simple right shift by p: `0x2AA000` becomes `0x2AA0`, which fits perfectly in 16 bits and, when divided by 2^8, yields again 42.625. It must be noted that this shift operation is not immune to precision loss if the fractional part of the multiplication requires more than p bits to be represented.

Opposite to the multiplication, the division between x and y results in:

$$\frac{x}{y} = \frac{n_1 \cdot 2^{-p}}{n_2 \cdot 2^{-p}} = \frac{n_1}{n_2} \tag{4.6}$$

The operation requires a corrective factor as well, that is a multiplication by 2^{-p}. However, the division presents an additional issue with respect to the multiplication: in equation 4.6, the fractional bits are completely lost. Even if the corrective factor 2^{-p} is applied at the end of the computation, the fractional part would be composed of all 0s. To avoid information loss, it is possible to apply 2^{-p} *before* the division takes place as:

$$\frac{n_1 \cdot 2^{-p} \cdot 2^{-p}}{n_2 \cdot 2^{-p}} = \frac{n_1}{n_2} \cdot 2^{-p} \tag{4.7}$$

Which produces the correct result in the expected format.

4.3.2 Kernel Library

Some of the mathematical transformations described in 4.4.1 operates in $\mathbb{R}$. However, the usage of floating point numbers is discouraged in kernel code [18] for performance and compatibility reasons, therefore kernel developers resort to fixed point for managing these kinds of quantities. The kernel does provide some functions to abstract the calculations described in section 4.3.1, but they are scattered through various compilation units and, in general, lacks in functionalities. Part of the work in this thesis dealt with the development of a small fixed point arithmetics library to provide a consistent kernel interface and proper support to fixed point operations. As of now, the function set is restricted to the need of this thesis, which uses 10 bits to represent the fraction, that is:

- `uint_to_ufix()`: converts a 64-bit unsigned integer to fixed point number. It also takes care of possible overflow errors, throwing away the least significant bits to maintain the most significant ones, reducing the effects of information loss.
- `sint_to_sfix()`: it is the same as `uint_to_ufix()`, but for signed numbers.
- `sfix_to_sint()`: converts a signed fixed point number to a signed 64-bit integer. It implements a basic rounding scheme to reduce error accumulation.
- `fix_udiv_64()`: divides two unsigned fixed point numbers. The return value is also a fixed point number on 64 bits.
- `fix_smul_64()`: multiplies two signed fixed point numbers. If the kernel is compiled with support for 128 bits integers, the multiplication output is temporarily stored on 128 bits to reduce the probability of overflow, otherwise it resorts to standard 64-bit variables. The return value is a signed fixed point number.
- `fix_umul_64()`: it is the same as `fix_smul_64()`, but operates on unsigned values.

4.4 Scheduler Integration

This section details the main contributions of the thesis. The CFS scheduler is extended to introduce a penalty/reward score system applied at the task level. The score represent the global intensiveness of the thread on the monitored resources as a percentage. It is a multiplicative factor increasing or decreasing key CFS quantities such as the *virtual runtime* and the *timeslice* of a thread. Penalizing a thread means increasing the `vruntime`, pushing it farther in the execution timeline to reduce its chances to be scheduled, and decreasing the timeslice, reducing the amount of expendable time on the core.

To perform a comprehensive analysis, various kernel configurations have been generated: a first iteration considers only the PMCs related to cache events to compute the *score*. A second iteration, instead, combines the caches and branch predictor PMCs with different *weights* to study the best configuration. The *score* is also reversed in some of the kernel versions to understand if threads must be penalized or rewarded when they put under heavy stress the hardware resources.

4.4.1 Mathematical Model

Let $cm(t)$ and $ca(t)$ be the number of cache misses and cache accesses at time t. It is possible to compute the cache miss rate $cmr(t)$ as:

$$cmr(t) = \frac{cm(t) - cm(t-1)}{ca(t) - ca(t-1)} = \frac{\Delta_{cm}}{\Delta_{ca}} \tag{4.8}$$

From this definition, is always true that:

$$0 \leq cmr(t) \leq 1$$

The branch miss rate $brm(t)$ can be derived from equation 4.8 by replacing cm with the number of branch mispredictions bm and ca with the number of branch instruction executed be.

The PMCs combination $c(t)$, representing how much intensive is a thread on the monitored resources, is computed differently when only caches are being monitored (equation 4.9), or also branch misses are taken into account (equation 4.10):

$$c(t) = cmr(t) \tag{4.9}$$
$$c(t) = w_1 \cdot cmr(t) + w_2 \cdot bmr(t) \tag{4.10}$$

With w_1 and w_2 being the rate's weights. In this study, three different weights set have been considered:

- $w_1 = w_2 = 0.5$: the cache miss rate and the branch misprediction rate are weighted equally.
- $w_1 = 0.75,\ w_2 = 0.25$: the cache miss rate is more important than the branch misprediction rate.
- $w_1 = 0.25,\ w_2 = 0.75$: the branch misprediction rate is more important than the cache miss rate.

By construction, also for $c(t)$ is always true that:

$$0 \leq c(t) \leq 1$$

In its current form, $c(t)$ is not still usable in CFS: as stated above, the *score* is multiplied to scheduling quantities to increase or reduce their values. A multiplication with $c(t)$, instead, always results in a bigger value. Hence, the *score* $s(t)$ is derived as:

$$s(t) = c(t) - 0.5 \tag{4.11}$$

Where $s(t) \in [-0.5, 0.5]$. The boundary values have been chosen to not over- penalize or reward a thread, compromising excessively CFS's fairness. The *penalty* is finally computed as an exponential moving average:

$$penalty(t) = \alpha \cdot s(t) + (1 - \alpha) \cdot penalty(t - 1) \tag{4.12}$$

The exponential average has the nice property to smooth out short-term fluctuations, hence it represents the *global* thread behavior. In the general formulation $\alpha \in [0, 1]$, but in this thesis it is fixed to 0.5 to limit the already wide number of kernel configurations to test. It is clear that:

$$\forall\ \alpha \Rightarrow penalty(t) \in [-0.5, 0.5] \tag{4.13}$$

In equation 4.12, a positive $penalty(t)$ value means that a thread shall be *penalized*. Conversely, a negative quantity means that it shall be *rewarded*. With the current formulation, a thread is penalized when $c(t) > 0.5$ consistently, that is, the content of the monitored resources is being trashed. As stated earlier, it has been studied also the case where threads triggering many cache misses or branch mispredictions shall be rewarded. It is easy to extend this model to comprehend such case by reversing $c(t)$ as:

$$c^{\star}(t) = 1 - c(t) \tag{4.14}$$

$c^{\star}(t)$ substitutes $c(t)$ in equation 4.11 and the rest of the formulation holds. In this situation, a thread with a high $cmr(t)$ and $brm(t)$ will have a low $c^{\star}(t)$ value, consequently $penalty(t) < 0$ and a *reward* will be given.

$penalty(t)$ is the multiplicative factor applied to the *virtual runtime* and *timeslice* of a task to modify the scheduler behavior.

Virtual runtime

Consider equation 3.3. $penalty(t)$ cannot be directly applied to the final $vruntime(t)$ value, because in case of a reward the resulting value would be smaller than the original and, by definition, $vruntime(t)$ is a monotonically increasing quantity. However, the equation is composed of two operands: the cumulative virtual runtime, computed in previous steps, and the current CPU execution time weighted by the task's priority. The second operand is the $vruntime$ spent in execution by the thread between two scheduler invocations. Equation 3.3 can be reformulated as:

$$vruntime(t) = vruntime(t-1) + \Delta_{fair} \tag{4.15}$$

Where:

$$\Delta_{fair} = \Delta_{exec} \cdot \frac{NICE\_0\_LOAD}{curr \rightarrow load} \tag{4.16}$$

$penalty(t)$ can be multiplied to equation 4.16 to derive a quantity to increase or decrease the value of Δ_{fair} itself, thus changing the increase rate of $vruntime(t)$, as:

$$\Delta^{\star}_{fair} = \Delta_{fair} + \Delta_{fair} \cdot penalty(t) \tag{4.17}$$

$$vruntime(t) = vruntime(t-1) + \Delta^{\star}_{fair} \tag{4.18}$$

It is worth to analyze what happens to $\Delta^{\star}_{fair}$ with respect to Δ_{fair} based on $penalty(t)$'s sign:

- $penalty(t) < 0$: $\Delta^{\star}_{fair} < \Delta_{fair}$. $vruntime(t)$ progresses in a slower way when compared to the stock kernel. It means that the thread will be placed toward the left side of the red-black tree, increasing its chances to be rescheduled soon.
- $penalty(t) = 0$: $\Delta^{\star}_{fair} = \Delta_{fair}$. The behavior is the same as the stock kernel.
- $penalty(t) > 0$: $\Delta^{\star}_{fair} > \Delta_{fair}$. $vruntime(t)$ increases at a faster pace with respect to the stock kernel. The thread will be placed toward the right side of the red-black tree, therefore its scheduling opportunities will diminish.

Timeslice

Let Δ_{slice} be:

$$\Delta_{slice} = timeslice \cdot penalty(t) \tag{4.19}$$

Where $timeslice$ is the result of equation 3.5. Δ_{slice} is the amount to be *subtracted* from $timeslice$ to augment or reduce the CPU time assigned to the thread in proportion with $penalty(t)$. Essentially, the timeslice calculation is modified as:

$$timeslice^{\star} = timeslice - \Delta_{slice}. \tag{4.20}$$

To understand why a subtraction is used, equation 4.20 can be analyzed in function of $penalty(t)$'s sign:

- $penalty(t) < 0$: The thread must be rewarded and its timeslice must increase. In this case $\Delta_{slice} < 0$, hence with the subtraction $timeslice^{\star} > timeslice$.
- $penalty(t) = 0$: The timeslice must be the same as in the stock version. In fact, $\Delta_{slice} = 0$ and $timeslice^{\star} = timeslice$.
- $penalty > 0$: The thread is penalized, hence the desired outcome is $timeslice^{\star} < timeslice$. In this case $\Delta_{slice} > 0$, so the subtraction leads to a smaller timeslice.

4.4.2 Model Implementation

This section presents the modifications required to integrate PMC-based scheduling in the CFS. The scheduler resorts to the PMC library presented in section 4.2 to read performance counter values, that feed the theoretical model discussed in section 4.4.1 relying on the fixed point library detailed in section 4.3 to manage rational quantities. The code shown in this section is the one strictly necessary to integrate PMCs within the scheduler. There is more dedicated to logging purposes, discussed in section 4.5.

Task Struct Members

The scheduler requires a minimal amount of additional memory in every task to save its state. In particular, each task in the system holds its penalty and the PMCs state to compute the difference in equation 4.8. Therefore, `struct task_struct` becomes:

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    struct thread_info  thread_info;
#endif

    ...

#ifdef CONFIG_PMC_SCHED
    u64                 start_pmcs[PMCS_USED];
    s64                 penalty;
#endif

    struct thread_struct        thread;
};
```

Listing 4.3: struct task_struct additions

`start_pmcs` is an array storing the second operand of equation 4.8 for every performance counter in use. `penalty`, instead, corresponds to $penalty(t)$ in equation 4.12. These struct members have been placed in `task_struct`, and not `sched_entity`, to ease the extension to more scheduling classes in future works. The total memory required by these additions varies with the kernel configuration in use. In this thesis, valid `PMC_USED` values are 2, 3 and 4, resulting in per-task memory overhead of 24, 32, or 40 bytes.

Special consideration is needed when managing `penalty`: even if it is declared as a signed 64 bits integer, the code uses it as a *signed fixed point* number with the 10 least significant bits serving as the fractional part.

The variables are initialized in `__sched_fork()`, a function called by `copy_process()` during the main `clone()` system call routine, to set up scheduling related parameters to their default values:

```
static void __sched_fork(unsigned long clone_flags, struct task_struct
    *p)
```

```
{
    p->on_rq = 0;

    ...

#ifdef CONFIG_PMC_SCHED
    memset(p->start_pmcs, 0, sizeof(u64) * PMCS_USED);
    p->penalty = 0;
#endif

    ...
}
```

Listing 4.4: __sched_fork initialization

PMCs update

The scheduler needs to periodically access performance counters to evaluate threads-hardware interactions and update the relatives penalties. For this purpose, the main scheduling routine `__schedule()` has been modified to call `update_pmcs()`, shown in listing 4.5, to collect the statistics of the task being switched out and set up the parameters of the one being switched in. Evaluating performance counters exclusively in `__schedule()` allows to monitor also the kernel's system calls impact, resulting in a complete task profile, improving the quality of the collected data. `update_pmcs()` is invoked just before the kernel determines if `next` and `prev` are different tasks in listing 3.5.

```
static void update_pmcs(struct task_struct *prev, struct task_struct *
    next)
{
    u64 *start_pmcs = prev->start_pmcs;
    u64 end_pmcs[PMCS_USED];
    u64 end_tot_load_uops;
    u64 start_tot_load_uops;
    u64 cm_rate;
    u64 bm_rate = 0;
    int pmc;

    for (pmc = 0; pmc < PMCS_USED; pmc++) {
        end_pmcs[pmc] = read_pmc(pmc);
    }

    end_tot_load_uops = end_pmcs[PMC0] + end_pmcs[PMC1];
    start_tot_load_uops = start_pmcs[PMC0] + start_pmcs[PMC1];

#if defined(CONFIG_PMC_L1) && defined(CONFIG_PMC_CACHE_ONLY)
    end_tot_load_uops += end_pmcs[PMC2];
    start_tot_load_uops += start_pmcs[PMC2];
    compute_rate(cm_rate,
```

```
22                  (end_pmcs[PMC1] + end_pmcs[PMC2]) - (start_pmcs[PMC1]
    + start_pmcs[PMC2]),
23                  end_tot_load_uops - start_tot_load_uops,
24                  prev->normalized_cm_rate);
25 #else
26      compute_rate(cm_rate,
27                  end_pmcs[PMC1] - start_pmcs[PMC1],
28                  end_tot_load_uops - start_tot_load_uops,
29                  prev->normalized_cm_rate);
30 #endif
31 #ifndef CONFIG_PMC_CACHE_ONLY
32      compute_rate(bm_rate,
33                  end_pmcs[PMC3] - start_pmcs[PMC3],
34                  end_pmcs[PMC2] - start_pmcs[PMC2],
35                  prev->normalized_bm_rate);
36      bm_rate = fix_umul_64(bm_rate, BRM_SCALE_FACTOR);
37      if (bm_rate > POS_FIX_1) {
38          bm_rate = POS_FIX_1;
39          prev->bm_rate_acc++;
40      }
41 #endif
42
43 #ifndef CONFIG_PMC_LOG_ONLY
44      compute_penalty(prev, cm_rate, bm_rate);
45 #endif
46
47      for (pmc = 0; pmc < PMCS_USED; pmc++) {
48          next->start_pmcs[pmc] = end_pmcs[pmc];
49      }
50 }
```

Listing 4.5: update_pmcs

Before starting the explanation, the PMC configuration used in this thesis is shown in table 4.1.

Index	PMC name	Kernel configurations
0	MEM_LOAD_UOPS_RETIRED.Lx_HIT	All
1	MEM_LOAD_UOPS_RETIRED.Lx_MISS	All
2	MEM_LOAD_UOPS_RETIRED.HIT_LFB	L1 cache monitored, BPU ignored
2	Branch Instruction Retired	Cache and BPU monitored
3	Branch Misses Retired	Cache and BPU monitored

Table 4.1: PMC index - event mapping

At the beginning of the function, the performance counters values are read and stored in `end_pmcs`. Cache performance counters provide the count for hit and miss events, however the cache miss rate is defined as $\frac{cache\ miss}{cache\ accesses}$. Hence, the total number of load μops is obtained by summing cache hits and misses. `end_tot_load_uops`

corresponds to $ca(t)$ in equation 4.8, `start_tot_load_uops` to $ca(t-1)$. The code carries the cache miss rate computation in two different ways depending on the kernel configuration: if the cache L1 is the only hardware resource monitored, it accounts also the line fill buffer events in the total amount of cache accesses. Then, with the macro `compute_rate()`, defined in listing 4.6, it calculates the cache miss rate as in equation 4.8 with the cleverness of adding to the cache misses the number of line fill requests.

On the other hand, if a different cache from the L1 is being monitored, or the BPU events are taken into consideration as well, the cache miss rate derivation requires no additional correction and the function directly applies equation 4.8 to the collected values.

```
#define compute_rate(rate, num, den, norm)  \
do {                                        \
    (rate) = fix_udiv_64((num), (den));     \
    if (unlikely((rate) > POS_FIX_1)) {     \
        (rate) = POS_FIX_1;                 \
        (norm)++;                           \
    }                                       \
}while (0)
```

Listing 4.6: compute_rate macro

The macro exploits `fix_udiv_64()` to compute the fixed point division between `num` and `den`. Even if these parameters are not stored in fixed point, the calculation is nevertheless correct. In fact, the division outcome would be the same as in equation 4.6, but since the function applies the corrective factor before the division takes place, the outcome is a well-formed fixed point value. This is a small micro-optimization since no conversion is required. If $den > num$ there is an error in the parameters, hence the macro normalizes the rate to `POS_FIX_1`, a constant value representing the fixed point number corresponding to 1, and raises an error flag used during debug. For more details about debugging and logging refer to section 4.5.

If BPU events are being monitored, that is, `CONFIG_PMC_CACHE_ONLY` is not defined, the function uses `compute_rate()` to compute the branch miss rate as well. Branch predictors usually have an excellent hit rate: already in the first half of 1990 there exist prediction schemes capable of correctly guessing the 90 % of branch outcomes [47]. Research has since improved the hit rate to values higher than 95 % [3, 25]. Therefore, the branch miss rate is scaled by 10 to pull out usable quantities in equation 4.10. Due to time constraints, a full characterization of the branch predictor has not been possible, but from the above discussion a multiplication by 10 is a conservative solution.

The miss rates, along with the pointer to the task being scheduled out, are passed as parameters to `compute_penalty()` for the penalty calculation. Then, the code updates the `start_pmcs` member of the task being scheduled in with the values previously read. This operation must be necessarily performed at the end, because it may be possible that `prev` and `next` points to the same task. If `next`'s update were executed earlier, in case `next == prev` the difference of `end_pmcs` and `start_mpcs` would be 0. Consequently,

the rates would be 0 as well, defeating the purpose of the penalty/reward mechanism.

Penalty Calculation

`compute_penalty()` is the function in charge of computing $penalty(t)$, and it is shown in listing 4.7. It supports many compile-time options to alter the calculation process and allow various experiments.

```
static void compute_penalty(struct task_struct *t, u64 cm_rate, u64
    bm_rate)
{
    u64 c;
#if defined(CONFIG_PMC_EQUAL_WEIGHT)
    c = fix_umul_64(cm_rate, M) + fix_umul_64(bm_rate, M);
#elif defined(CONFIG_PMC_CACHE_WEIGHT)
    c = fix_umul_64(cm_rate, M) + fix_umul_64(bm_rate, S);
#elif defined(CONFIG_PMC_BRANCH_WEIGHT)
    c = fix_umul_64(cm_rate, S) + fix_umul_64(bm_rate, M);
#else /* only cache miss are in use */
    (void)bm_rate;
    c = cm_rate;
#endif

#ifndef CONFIG_PMC_CACHE_ONLY
    if (unlikely(c > POS_FIX_1)) {
        t->normalized_combination++;
        c = POS_FIX_1;
    }
#endif

#ifndef CONFIG_PMC_HS_PENALTY
    c = POS_FIX_1 - c;
#endif
    c = c - POS_FIX_05;

    t->penalty = fix_smul_64((s64)c, ALPHA) + fix_smul_64(t->penalty,
    BETA);
    if (unlikely(t->penalty > POS_FIX_05)) {
        t->normalized_penalty++;
        t->penalty = POS_FIX_05;
    } else if (unlikely(t->penalty < NEG_FIX_05)) {
        t->normalized_penalty++;
        t->penalty = NEG_FIX_05;
    }
}
```

Listing 4.7: compute_penalty

The PMC combination $c(t)$ is computed differently based on the kernel configuration. If `CONFIG_PMC_CACHE_ONLY` is set $c(t)$ simply correspond the cache miss rate, as in equation 4.9. Otherwise, the code resorts to equation 4.10, setting the weights to:

- `CONFIG_PMC_EQUAL_WEIGHT` is defined: $w_1 = w_2 = M = 512$, where 512 corresponds to 0.5 in the fixed point format adopted.
- `CONFIG_PMC_CACHE_WEIGHT` is defined: $w_1 = M = 768$ and $w_2 = S = 256$, where 768 and 256 are the fixed point values coinciding with 0.75 and 0.25.
- `CONFIG_PMC_BRANCH_WEIGHT` is defined: $w_1 = S = 256$ and $w_2 = M = 768$.

Once the function computes $c(t)$, it applies the usual normalization step common to most of the previous calculations only if both the cache and the BPU events are considered. In fact, in the case where only the caches are monitored, the normalization step is already executed by `compute_rate()` in `update_pmcs()`.

`CONFIG_PMC_HS_PENALTY` is the macro acting as a toggle to select at compile-time the penalty or reward algorithm. If it is not set, $c(t)$ is inverted as in equation 4.14, hence threads generating many cache or branch misses are *rewarded*. On the other hand, if `CONFIG_PMC_HS_PENALTY` is defined, equation 4.14 is not compiled and heavy threads are *penalized*.

Eventually, the code updates the `penalty` of the task being scheduled out by applying equation 4.12 and, in case it is out of the range defined in 4.13, it normalizes the value and increases the related debug flag. Since `penalty` is updated only when a task is leaving the CPU, tasks short enough to complete their execution in a single scheduling period are exempted from the algorithm's effect.

PMC-based Scheduling

With the core scheduler able to interact with performance counters and to compute the `penalty`, it is now possible to discuss the CFS modifications required to enable PMC-based scheduling. One of the key parameters is the *virtual runtime*, detailed in section 3.3.4, which is updated in `update_curr()`. The function requires minimal modifications to transform the calculation of equation 3.3 in equation 4.15, as shown in listing 4.8:

```
1  static void update_curr(struct cfs_rq *cfs_rq)
2  {
3      struct sched_entity *curr = cfs_rq->curr;
4      u64 now = rq_clock_task(rq_of(cfs_rq));
5      u64 delta_exec;
6  #if defined(CONFIG_PMC_SCHED) && !defined(CONFIG_PMC_LOG_ONLY)
7      u64 delta_fair;
8      s64 delta_fair_star;
9      struct task_struct *task;
10 #endif
11
12     if (unlikely(!curr))
13         return;
14
```

```
15      delta_exec = now - curr->exec_start;
16      if (unlikely((s64)delta_exec <= 0))
17          return;
18
19      curr->exec_start = now;
20
21      schedstat_set(curr->statistics.exec_max,
22                    max(delta_exec, curr->statistics.exec_max));
23
24      curr->sum_exec_runtime += delta_exec;
25      schedstat_add(cfs_rq->exec_clock, delta_exec);
26  #if defined(CONFIG_PMC_SCHED) && !defined(CONFIG_PMC_LOG_ONLY)
27      delta_fair = calc_delta_fair(delta_exec, curr);
28      if (entity_is_task(curr)) {
29          task = task_of(curr);
30          delta_fair_star = fix_smul_64(uint_to_ufix(delta_fair), task->
        penalty);
31          delta_fair_star = sfix_to_sint(delta_fair_star);
32          delta_fair += delta_fair_star;
33      }
34
35      curr->vruntime += delta_fair;
36  #else
37      curr->vruntime += calc_delta_fair(delta_exec, curr);
38  #endif
39      ...
40  }
```

Listing 4.8: update_curr modifications

`calc_delta_fair()`'s return value is not directly applied to `vruntime` anymore, but it is stored in a variable called `delta_fair`. Then, the function checks if `curr` is a scheduling entity associated to a task through `entity_is_task()`, an utility macro detecting if a scheduling entity is also a task. The condition is very simple: an entity is a task if it does not own a runqueue, hence the macro expands to `!curr->my_rq`. As performance counters can be evaluated only on actual threads, and not on abstract entities such as tasks groups, the code does not alter CFS behavior if `curr` is not a task. On the other hand, if the scheduling entity is related to a task, it retrieves the pointer to the task containing `curr`. This step is necessary because `penalty` is stored in the process descriptor, that is `task_struct`. Subsequently, the function saves in `delta_fair_star` the proportion based on `penalty` to increase/decrease `delta_fair` using a signed fixed point multiplication on 64 bits. Since `delta_fair` is not stored in a fixed point, before the multiplication takes place it is necessary to convert the number to a fixed point representation with `uint_to_ufix()`. Performing the multiplication in fixed point allows the scheduler to manage values smaller than 1, however CFS operates mainly on integer quantities and `vruntime` is no exception. Therefore, `sfix_to_sint()` converts `delta_fair_star` back to the original integer representation and it is added to `delta_fair`, which is in turn added to the `vruntime`.

The other CFS key parameter, *timeslice*, is updated in a similar manner as `vruntime`. Listing 3.15 and 4.9 can be compared to appreciate again the small number of modifications required to introduce PMCs in the CFS.

```
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
#if defined(CONFIG_PMC_SCHED) && !defined(CONFIG_PMC_LOG_ONLY)
    struct task_struct *task;
    s64 delta_slice;
    struct sched_entity *orig_se = se;
#endif
    u64 slice = __sched_period(cfs_rq->nr_running + !se->on_rq);

    for_each_sched_entity(se) {
        struct load_weight *load;
        struct load_weight lw;

        cfs_rq = cfs_rq_of(se);
        load = &cfs_rq->load;

        if (unlikely(!se->on_rq)) {
            lw = cfs_rq->load;

            update_load_add(&lw, se->load.weight);
            load = &lw;
        }
        slice = __calc_delta(slice, se->load.weight, load);
    }

#if defined(CONFIG_PMC_SCHED) && !defined(CONFIG_PMC_LOG_ONLY)
    if (entity_is_task(orig_se)) {
        task = task_of(orig_se);
        delta_slice = fix_smul_64(uint_to_ufix(slice), task->penalty);
        delta_slice = sfix_to_sint(delta_slice);
        slice -= delta_slice;
    }
#endif
    return slice;
```

Listing 4.9: sched_slice modifications

With respect to `update_curr()`, there is an additional local variable called `orig_se`. It is a `sched_entity` pointer initialized to `se`. `orig_se` is fundamental to ensure the correctness of operations because, in the original timeslice calculation, the loop continously update the value of `se`, which eventually becomes NULL. Once the original timeslice calculation ends, if the original scheduling entity is also a task, the code computes `delta_slice` as in equation 4.19. Similarly to `delta_fair`, `delta_slice` must be converted back to a standard integer representation before it can be applied to the timeslice. At the end, the function derives the PMC-based timeslice by *subtracting* `delta_slice` from `slice` and returns such quantity to the caller.

4.5 Data Collection

To measure the scheduler impact on performance and to verify the correctess of operations, some data must be collected and analyzed. For security reasons, the kernel does not provide a way to access its internal structures from userspace. However, Linux exposes few logging interfaces, detailed in section 4.5.1, to obtain a selected amount of information about the inner systems. This work exploits them to provide useful data for debugging and benchmarking. However, kernel logging is an expensive operation, thus it must be reduced to a bare minimum. In user-space, a custom C program called `logger` runs during the benchmarks to collect log messages and save them on disk for offline processing.

4.5.1 Kernel Logging

Kernel logging is a fundamental operation for benchmarking and debugging. It is possible to accurately measure the execution time of a task without the latency overhead intrinsically present in system calls such as `wait()`, or to output a vast amount of information not accessible by userspace tools.

Logging Mechanism

The kernel provides multiple frameworks for logging, the most famous one being `printk()` [32]. However, `printk()` has a non-negligible overhead, which makes it unsuitable for performance critical paths like the scheduler. For these particular situations, the kernel offers an alternative framework called Ftrace [9]. Ftrace supports a wide range of tools, known as *tracers*, to measure system latency, performance, call stacks and so on. It also offers a macro, `trace_printk()`, which uses a printf-like interface to log messages in a dedicated, per-CPU, ring buffer. A log line produced by Ftrace looks like:

```
systemd-1 [000] .... 0.124413: copy_process: Process pid 0 forked
```

Where:

- `systemd` is the name of the task in execution during a `trace_printk()` call.
- `1` is the pid of the task currently running on the core. As explained in section 3.1.2, this pid refers to the task's unique identifier, not to the userspace homonymous.
- `[000]` indicates the core number executing the `trace_printk()` call.
- `....` are flags specifying a portion of the system context. In order, from left to right, they are: `irqs-off`, `need-resched`, `hardirq/softirq` and `preempt-depth`.

- `0.124413` is the log timestamp. The format is "seconds.microseconds" and is measured since boot.
- `copy_process` is the kernel function containing the `trace_printk()` call.
- `Process pid 0 forked` is the log message.

Ftrace creates an entry in the Debugfs file system, usually located at `/sys/kernel/debug/tracing`, to interact with tracers. `trace_printk()` is accessible from two different files: `trace` and `trace_pipe`. The difference is that the former does not *consume* the ring buffer content, hence if tracing is stopped through the API, executing `cat trace` will always output the same data. The latter, on the other hand, is meant for *live tracing* and consumes the content. If the ring buffer is empty, either because no `trace_printk()` has been executed or because everything was read, read operations will *block* until new data is available [14].

Logging Process

As stated earlier, the kernel logs messages for both benchmarking and debugging purposes. From a purely benchmarking perspective, the measure of interest is the total tasks' execution time, that is, the amount of time elapsed between process creation and death. Within the kernel, it is possible to collect precise timing information about these two instant, not only for processes, but also for individual threads: as explained in section 3.1.2, `fork()` and `pthread_create()` resort to the `clone()` syscall for creating new processes and threads, which eventually lead to the execution of `copy_process()`. Once a task ends its work, whether it is a thread or a process, the kernel invokes `do_exit()` to signal the task's death. Additionally, for the first thread of the process, `do_group_exit()` is called as well. The difference between the moments in which the kernel calls `copy_process()` and `do_[group_]exit()` is an accurate representation of the tasks' execution time that consider also scheduling delays. The benchmark's data collection flow is shown in figure 4.3.

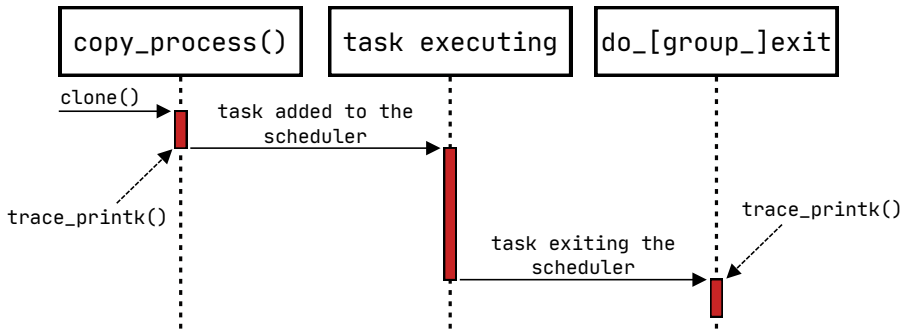


Figure 4.3: Benchmark data collection flow

The kernel patch adds two `trace_printk()` calls, one in `copy_process()` and one `do_exit()`. The former is placed at the end of the function as:

```
#ifdef CONFIG_PMC_SCHED
trace_printk("[PMC_SCHED] Process pid %d tgid %d forked\n", p->pid,
             p->tgid);
#endif
```

Where `p` is the pointer to the newly created `task_struct`. Every message added by the patch contains a prefix, "`PMC_SCHED`", which identifies in the log file the strings belonging to this work. As it will be seen, the userspace logger expoits this feature to reduce the amount of log lines saved on disk.

In `do_exit()`, instead, the `trace_printk()` call is placed at the beginning of the function:

```
#ifdef CONFIG_PMC_SCHED
    trace_printk("[PMC_SCHED] Task exiting\n");
#endif
```

Not shown in 4.3 for compactness and generality, `trace_printk()` is also added at the beginning of `do_group_exit()` to discern whether the task exiting is a thread or a process:

```
#ifdef CONFIG_PMC_SCHED
    trace_printk("[PMC_SCHED] Process exiting\n");
#endif
```

The offline parser matches these prints using the task pid, and determines the execution time by subtracting from the timestamp of the `do_[group_]exit()` log the one of the `copy_process()` message. While for the exit messages the task pid is indirectly provided by `trace_printk()`, the same is not true for `copy_process()`: the task is currently being created at this moment in time, hence it is not the one running on the CPU. The `pid` actually corresponds to the parent's one, therefore the message in `copy_process()` explicitly provide the task's `pid` and the `tgid`.

For what concerns debugging, instead, there is a `trace_printk()` call in `free_task_struct()`. This function is called once the task is already dead for releasing the memory associated to `task_struct`. For this reason, also in this case is necessary to explicitly specify the `pid` and the `tgid` stored in the structure to be freed. The message content depends on the kernel's compilation settings, summarized in table 4.2. Altough it is possible to disable PMC-based scheduling, the messages shown in this section remains always active to allow comparisons between the various kernel configurations.

During the task's execution, the kernel accumulates these quantities in additional `task_struct` members, introduced specifically for this print. To reduce the chances of requiring additional cache lines to hold the structure, which in turn would result in performance degradation, the members in table 4.2 are conditionally excluded from the compilation.

	PMC-based scheduling		Standard CFS	
Parameter	Cache & branches	Cache	Cache & branches	Cache
Cache miss rate	✓	✓	✓	✓
Branch miss rate	✓		✓	
Penalty	✓	✓		
Voluntary context switches	✓	✓	✓	✓
Non-voluntary context switches	✓	✓	✓	✓
N. of `__schedule` calls	✓	✓	✓	✓
N. of normalized cache miss rates	✓	✓	✓	✓
N. of normalized branch miss rates	✓		✓	
N. of normalized combinations	✓			
N. of normalized penalty	✓	✓		

Table 4.2: Logged kernel parameters

4.5.2 Userspace Logging

Kernel logs must be collected and stored on disk. Altough in theory logs may be collected by running `cat trace_pipe`, this approach fails for multiple reasons: first, sending a signal to kill and stop `cat` results in the entire data collection being lost. Moreover, the kernel modifications introduced with this work may not be the only code calling `trace_printk()`, therefore the log size may increase because of unrelated messages. This quickly becomes a major concern for two motives: log files require a lot of space on disk and log parsing is a computationally expensive activity. However, `cat` does not offer filtering capabilities and its combination with `grep` or `awk` would be inefficient, as the log system must have a minimal footprint on CPU time and memory to be transparent during the benchmarking.

To fulfill these needs, a small C program has been developed. Whenever a new string is available for processing, the logger tokenizes it to efficiently determine:

- the process name and kernel pid;
- the core logging the message;
- the kernel function containing the `trace_printk()` call;
- the actual log message.

The tokenization process is performed in-place, hence there is no overhead due to dynamic memory allocation. In fact, the logger never allocates memory during parsing, but requests all the memory needed for operations before starting the parsing activity and frees it only when the it ends its execution.

Besides discarding messages not containing the "`[PMC_SCHED]`" preamble, it is possible to add custom filters on the process name, pid, the core number or the kernel function to further limit the file content, a feature especially useful for debugging purposes.

Chapter 5

Results

This chapter presents the adopted testing methodology and an analysis of the experimental results. Section 5.1 details the test infrastructure, consisting of two different compute- and memory- intensive workload to put the scheduler under heavy stress. Section 5.2 describes how the logs are processed to obtain relevant performance metrics to characterize the various scheduler versions. Finally, section 5.3 analyzes and discusses the result obtained in this work.

The test machine is equipped with an Intel i7-4720HQ CPU paired with 8 GiB of DDR3 RAM running Ubuntu 20.04 LTS.

5.1 Test Architecture

The scheduler performance is evaluated with a custom benchmark based on real-world applications. These belongs to two different groups: heavyweight and lightweight processes. The former contains multi-process and multi-thread programs performing complex task for a considerable amount of time, capable of putting under heavy stress the scheduler. The latter, instead, contains short-lived, single-thread and single-process programs introducing a perturbative action in the scheduler. Table 5.1 shows the list of heavyweight processes and table 5.2 the list of lightweight processes.

Name	Description
`Make` [12]	A utility for compiling C/C++ programs, paired with the `LLVM` infrastructure
`FFmpeg` [1]	A video transcoding software
`GNU Octave` [10]	A software for numerical computations

Table 5.1: Heavyweight processes

`Make`, when paired with the option `-jn` (with `n` being the number of available cores), becomes a multi-process program where each core processes a different compilation

unit. This processing is a control-intensive task, because the compiler must verify the correctness of the code against complex standards and often looks for optimization opportunities.

`FFmpeg` is instead a multi-thread application. Video transcoding is a heavy compute problem, so much that GPUs are often preferred to CPUs [19]. It also requires a high amount of memory to temporarily store uncompressed frames, hence the proper usage of caches is mandatory for acceptable performance.

`GNU Octave` uses a mix of processes and threads to complete its work. Numerical computations are intrinsically CPU demanding, and depending on the data set size also caches and memory usages may become a concern.

Name	Description
`AES-256` [36]	An efficient AES-256 algorithm implementation designed for embedded systems
`FFT` [17]	A program implementing the FFT algorithm and its inverse
`Susan` [17]	A lightweight image recognition package for embedded systems

Table 5.2: Lightweight processes

Lightweight processes' performance is not collected, as their only purpose is to introduce noise. They are computationally similar to (and hence emulate) tasks like mouse or keyboard interrupts, presentation of notifications, TCP/IP packet processing and so on.

Heavy processes execute two different workloads per kernel version, called *wl1* and *wl2* in the rest of this thesis. Table 5.3 summarizes their details.

	wl1	*wl2*
`Make`	C program compilation	C++ program compilation
`FFmpeg`	Transcode from 720p to 480p	Transcode from 1080p to 720p
`GNU Octave`	Matrices operations	Neural network training
Average execution time [s]	25	90

Table 5.3: Workload details

The workloads have been designed to analyze the patch's performance under different scenarios:

- *wl1* puts the system under stress and every monitored resource, i.e. caches and branch predictors, are extensively used.
- *wl2* is more intensive than *wl1* under every aspect. The C++ language complexity adds stress to the BPU, while transcoding a high resolution video and training a

neural network increase CPU and memory usage. Therefore, the number of cache misses becomes even more relevant to determine the tasks' total execution time.

5.1.1 Test Script

Figure 5.1 shows the steps implemented in a Python script to automate the kernel benchmark.

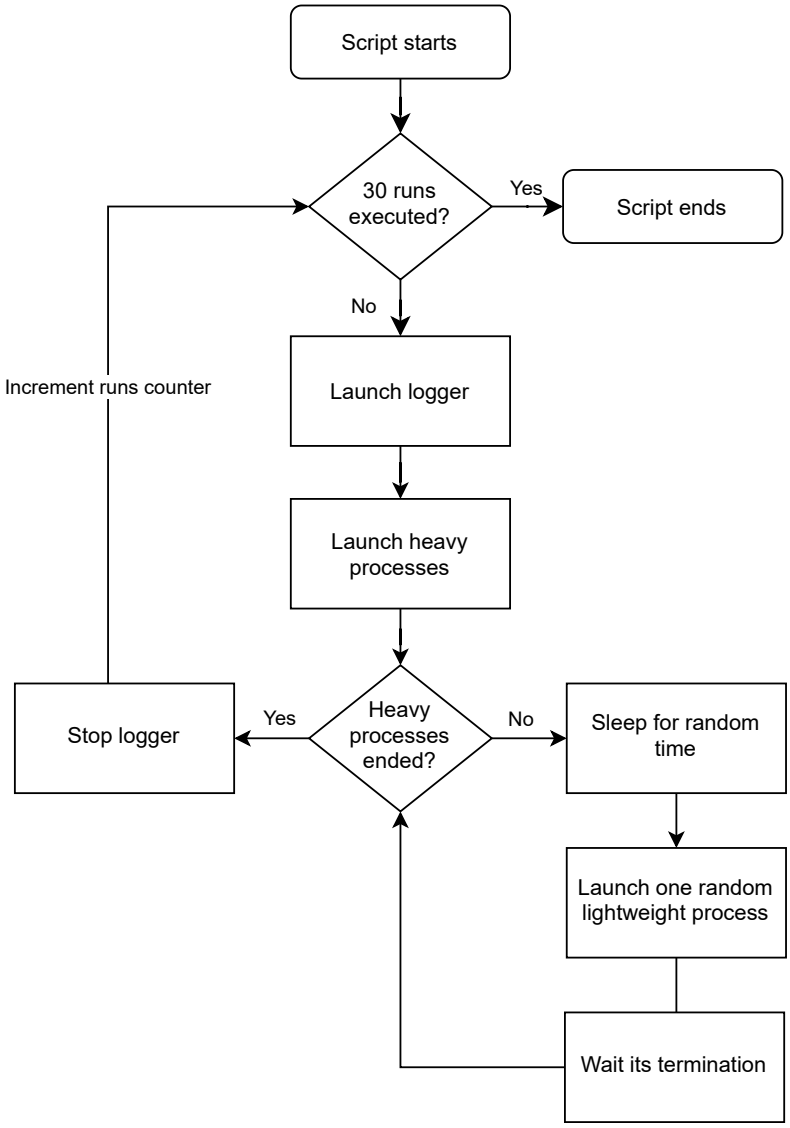


Figure 5.1: Test script flow

The script is run first for *wl1* and then for *wl2*. It starts the logger discussed in section 4.5.2 and the heavy processes in parallel. Then, until every heavy process has

not terminated its execution, the script goes to sleep for a variable amount of time comprised in the range $[500\ ms,\ 3000\ ms]$. Once it wakes up, the script spawns a random lightweight process from table 5.2 and waits for its completion. As soon as the heavyweight processes conclude their operations the logger is stopped. These steps are referred to as a *run*. The script executes 30 runs for each workload before returning control to the shell to provide statistical consistency.

Not shown in figure 5.1 for compactness, the test script creates a different folder for every kernel version to store the related logs and ease subsequent analysis, and produces a file for every run containing the user-space pids of the heavyweight processes, as it is needed by the log parser detailed in the next section.

5.2 Data Analysis

A set of Python scripts, that works in a hierarchical fashion, has been developed to parse and analyze the kernel logs collected with the test script. The first script processes the log file of a single *run*, taking as inputs the log file itself and the file containing the heavyweight processes' pids, called *root pids*. It builds a different tree for each heavyweight process and then merges the data of every tree to compute the execution time and the average miss rates. A second script launches the first script on every log file related to a particular kernel version and uses their outputs to compute the average, maximum, minimum and the standard deviation of the execution time, miss rates, context switches and so on of the heavyweight processes. Finally, a third script collects the output of the previous script to produce a variety of text files and plots comparing the results of the kernel patch against the baseline kernel.

The second and the third scripts do not perform complicated operations and are not interesting enough to be reproduced here, but it is worth to spend some time discussing how the first one works.

5.2.1 Log Parser

The script works in two distinct phases: in the first one it builds back the tasks' tree for each heavyweight process, then, once it reaches the end of the log file, it descends the tree to collect the data of every thread and produce global metrics.

Figure 5.2 shows the tree organization and the node's format:

- **pid**: the kernel pid, used as thread identifier in the tree.
- **tgid**: the thread group id.
- **name**: the thread's name, if any.
- **children**: the list of nodes, that is tasks, created by this thread.
- **start_time**: the instant in which the task was created, measured in microseconds.

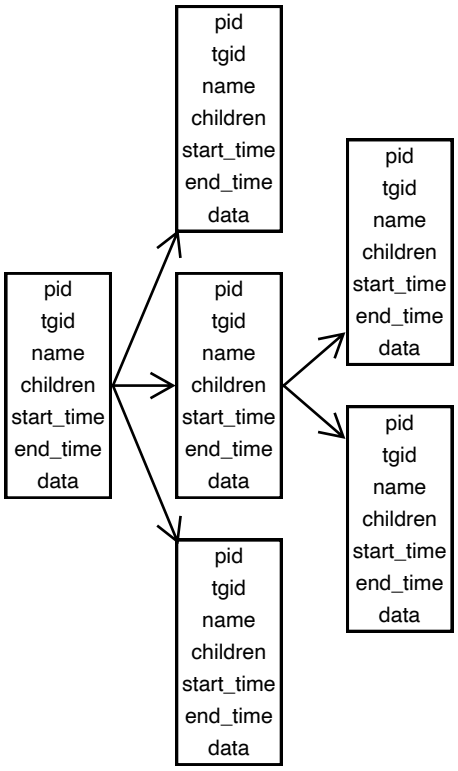


Figure 5.2: Parsing Tree

- **end_time**: the instant in which the task died, measured in microseconds.
- **data**: the task's cache miss rate, branch miss rate, number of context switches, etc.

The log parser reads one log line at the time, and based on the kernel function from which the message came, takes a different course of action. If the kernel function is `copy_process()`, it collects the pid and tgid contained in the message. First, it tries to match the pid with the *root pids*. If a correspondence is found, it creates the tree's root node setting `start_time` to the log timestamp. Otherwise, the script descends every tree until the pid added by `trace_printk()`, that is the parent's pid, matches the pid of a node. If such node is found, it creates a new node as before and adds it to the `children` list of the parent.

If a message comes either from `do_exit()` or `do_exit()`, the script walks the trees until it finds the node corresponding to the dying thread to update the `end_time` to the log line's timestamp.

Ultimately, if the message comes from `free_task()`, the parser searches the node matching the pid contained in the message and stores in `data` the kernel output according to table 4.2.

When the parser reaches the end of the log file it descends every tree, accumulating for each node the various statistics. Eventually, the accumulated cache miss rate, branch

miss rate and penalty are divided by the number of calls to `__schedule()` multiplied by 1024, that corresponds to 1 in the fixed point format adopted in this work, to compute the executable's global average values. In mathematical form:

$$global\ average = \frac{accumulated\ value}{nr\_schedule \cdot 1024} \tag{5.1}$$

The total execution time is calculated instead as:

$$execution\ time = end\_time_{root\ task} - start\_time_{root\ task} \tag{5.2}$$

Because the root task is always the first to start and the last to exit, since it must wait for any thread or process spawned to end. Once this information are obtained for every heavyweight process, the script returns the results to the caller script for further analysis.

5.3 Experimental Results

The results have been collected under two different configuration scenarios. In the first, presented in section 5.3.1, only cache-related events have been considered. Because of the limited number of available performance counters, each cache level has been evaluated in a different experiment. In the second, discussed in section 5.3.2, also the BPU-related events are monitored. As for the previous case, every cache level is analyzed independently. However, since the cache and branch PMCs can be combined with different weights, three different experiments are required for every cache level. It is important to make a disclaimer on the data quality: memory-related PMCs events on Intel 4th generation processors are affected by hardware bugs with no workaround. The errata in questions are HSM26 and HSM 30 [33].

5.3.1 Cache only

Figures 5.3 and 5.4 show for each cache level the speed-up of the kernels using the penalty/reward algorithms with respect to the original one. Tables 5.4 to 5.6 present the average cache miss rate, number of voluntary and number of non-voluntary context switches for every cache level. The "Original" column contains the data belonging to the original kernel, while "Penalty" and "Reward" show the percentage difference between the data of the penalty/reward algorithms and the original value. A negative *penalty* means that in average the task has been rewarded, a positive value means that it has been penalized.

For *wl1*, the plots show a clear pattern: the *reward* algorithm, which penalizes tasks with a low *score* and rewards task with a high *score*, consistently provide better performance. On the contrary, the *penalty* algorithm either provides no benefits, or reduces

the execution speed. The kernel version using the L1 cache PMCs is the one obtaining the most improvements from the reward algorithm, with Octave achieving a 9.35% speed-up and a general boost of 5.75% with respect to the original kernel.

Compared to Octave and FFmpeg, Make is the process feeling less the benefits of PMC-based scheduling. The reason lies in the different ways these applications exploit parallelism: FFmpeg and Octave use mainly long-lived threads to carry on their work, instead Make (and LLVM) relies on short-lived processes to compile the source code. These small processes may be even capable of compiling a single file in one scheduling period, avoiding the influence of the PMCs evaluations.

The speed-up obtained using the L2 and L3 cache misses has a different source compared to the one provided by the L1 PMCs. In the former case, the algorithm is effectively assigning a combination of penalties and rewards. Octave, the most cache-intensive process, must be allowed to possess the CPU for more time so that it can make better use of the cache. FFmpeg and Make, on the other hand, must be penalized to reduce cache utilization in favor of the processes that need it the most. In the latter case, instead, every process is heavily penalized. In fact, the L1 cache miss rate is always lower than 10%, therefore *penalty* is always higher than 40% and, consequently, the timeslice is almost halved. This behavior is reflected in the amount of non-voluntary context switches executed which, with the notable exception of Octave, increases significantly. Despite the additional overhead, the applications are clearly benefiting from a faster scheduling rate.

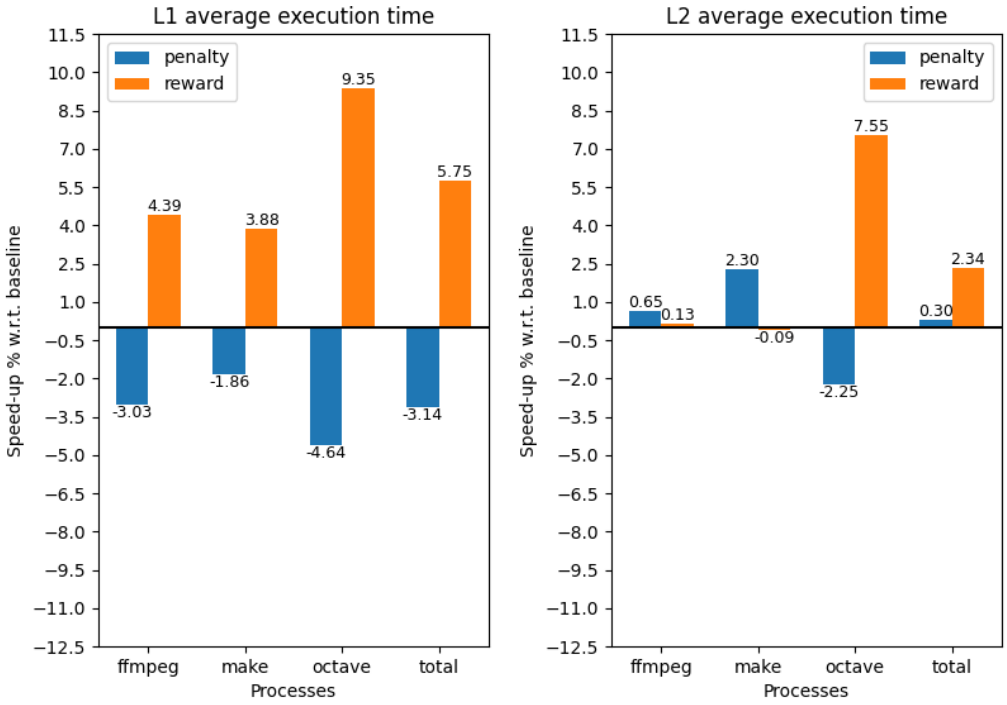
For *wl2*, the plots shows a much more complex situation. With the L1 PMCs, the reward algorithm still provides a speed-up - although more modest - but it performs worse than the original kernel with the L2 and L3 PMCs. In particular, with the L3 PMCs, it is the penalty algorithm to boost both local and global performances. Results between *wl1* and *wl2* are consistent only for the kernel using L1 PMCs and the reward algorithm, suggesting that tasks benefit from a higher scheduling turnaround, regardless of the cache activity.

		wl1			*wl2*		
		Original	Penalty (%)	Reward (%)	Original	Penalty (%)	Reward (%)
L1	CMR	9.661%	-1.487	-1.559	5.405%	-5.703	-0.518
	Penalty		-40.554	40.368		-44.944	44.482
	VCSW	46074	-0.498	0.485	8270	-10.238	3.240
	NVCSW	4587	-14.663	13.129	15440	-13.186	19.202
L2	CMR	48.226%	-1.632	1.261	43.020%	0.076	-0.630
	Penalty		-2.660	1.072		-7.053	7.164
	VCSW	46230	-1.115	0.614	8373	7.866	-7.265
	NVCSW	4663	3.470	-5.059	15687	4.830	-7.586
L3	CMR	45.458%	-0.849	4.347	25.936%	1.508	10.593
	Penalty		-5.007	2.455		-23.733	21.184
	VCSW	46136	-0.765	1.101	8395	-5.113	-4.400
	NVCSW	4711	-8.051	4.030	16651	-10.653	-6.509

Table 5.4: ffmpeg cache only data. CMR = cache miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches

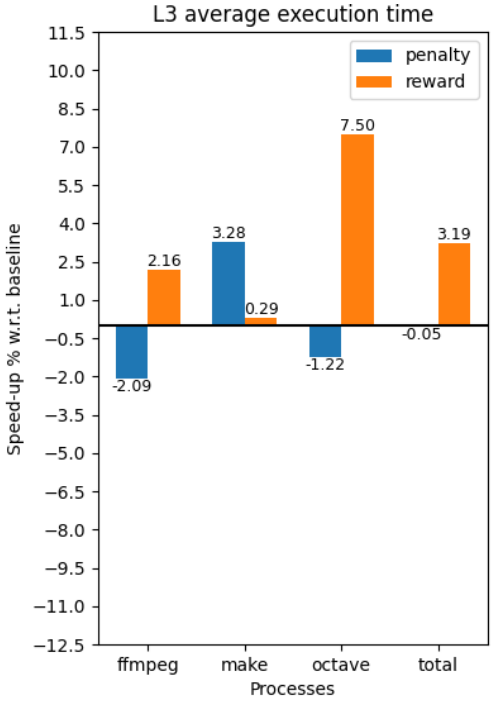
		wl1			*wl2*		
		Original	Penalty (%)	Reward (%)	Original	Penalty (%)	Reward (%)
L1	CMR	7.152%	4.547	-5.781	8.311%	3.696	-6.339
	Penalty		-40.741	41.916		-37.937	39.801
	VCSW	1263	-0.048	-2.338	10279	-0.551	0.389
	NVCSW	9932	-18.570	29.262	35884	-18.201	42.747
L2	CMR	42.417%	2.768	-1.648	45.908%	0.737	-2.594
	Penalty		-6.580	8.284		-4.004	5.400
	VCSW	1256	-1.681	-0.820	10268	0.048	0.260
	NVCSW	9716	-4.958	7.476	36922	-7.342	8.148
L3	CMR	29.974%	3.623	2.732	32.258%	4.806	-0.095
	Penalty		-18.784	19.010		-15.755	17.432
	VCSW	1242	1.152	0.226	10285	-0.503	0.374
	NVCSW	10048	-16.952	10.643	37498	-12.302	16.519

Table 5.5: make cache only data. CMR = cache miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches



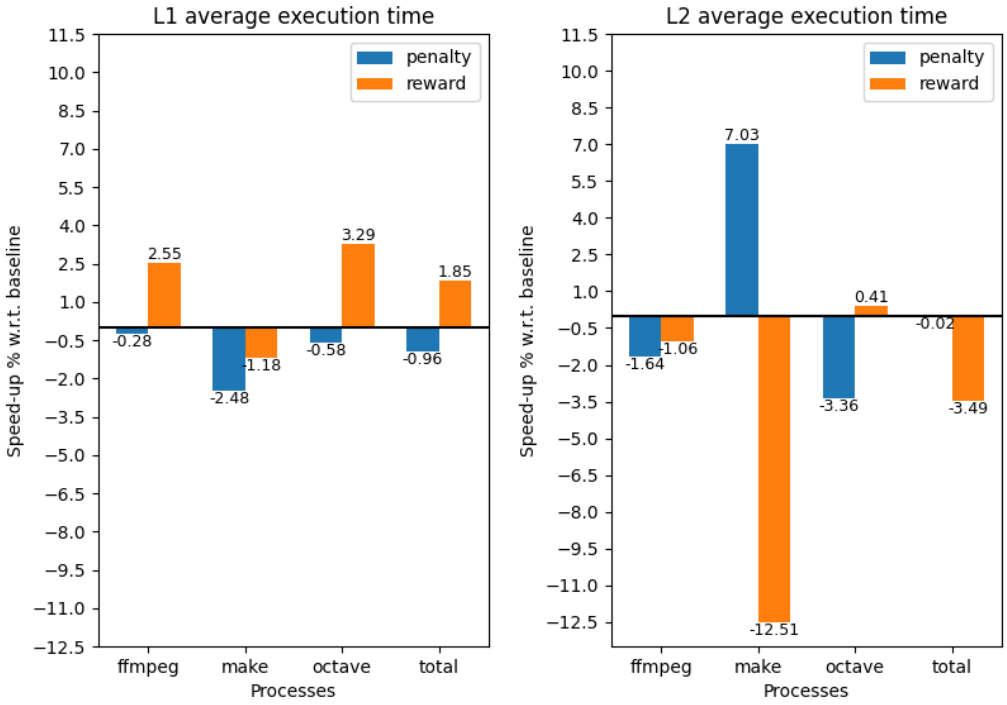
(a) L1

(b) L2



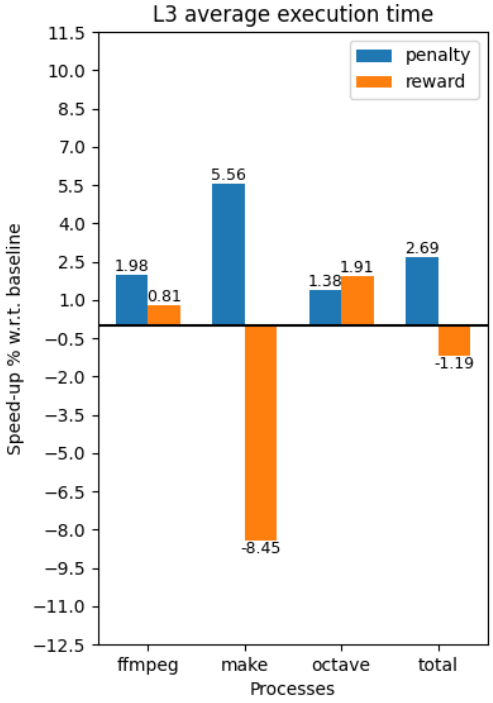
(c) L3

Figure 5.3: Cache PMCs wl1 results



(a) L1

(b) L2



(c) L3

Figure 5.4: Cache PMCs wl2 results

		wl1			*wl2*		
		Original	Penalty (%)	Reward (%)	Original	Penalty (%)	Reward (%)
L1	CMR	2.513%	-0.197	-2.407	1.774%	13.044	-6.448
	Penalty		-47.575	47.453		-48.088	48.229
	VCSW	259	-28.711	27.255	338	-23.700	5.516
	NVCSW	8414178	18.689	-29.114	59185755	10.775	-12.270
L2	CMR	66.558%	-9.996	5.415	70.375%	-7.448	-2.376
	Penalty		9.816	-20.236		15.042	-18.782
	VCSW	230	31.231	-4.947	332	-1.674	-11.891
	NVCSW	8722713	2.640	-25.447	60671201	0.659	-5.900
L3	CMR	55.657%	-3.261	10.956	39.155%	-9.828	24.692
	Penalty		3.783	-11.796		-14.734	1.128
	VCSW	248	-6.913	8.151	343	-8.632	-16.701
	NVCSW	8700104	9.451	-18.521	61658092	-0.132	0.943

Table 5.6: octave cache only data. CMR = cache miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches

5.3.2 Cache and Branches

Figures 5.5 and 5.6 show, for each cache level, the speed-up of the kernels using the penalty/reward algorithms and different weights parameters with respect to the original one. *equal-** are the kernels using the same weights for the cache and branch miss rates, *cache-** are the kernels weighting more the cache miss rate and *branch-** are the kernels weighting more the branch miss rate. Tables 5.7 to 5.12 present the average cache miss rate, average branch miss rate, number of voluntary and non-voluntary context switches for every cache level and algorithm. The algorithm names are *EQ* (equal), *CA* (cache) and *BR* (branch), which indicate the kind of weighting applied to the miss rates. It is important to outline once again that the branch miss rate presented in the tables and employed in the kernel is *scaled* by 10. Unlike the previous section, *wl1* and *wl2* are presented in two different tables due to page size constraints.

The results confirm what has been already discussed. For *wl1*, any reward algorithm is consistently better than the original kernel, while the opposite is true for the penalty algorithms. By observing the values of the non-voluntary context switches in tables 5.7 to 5.9 it is possible to appreciate how, especially in the L1 runs, the overhead of context switches is not directly correlated with performance. Due to the low value of the branch miss rates, *PMCs combination* $c(t)$ as computed in equation 4.10 almost never assumes a value higher than 0.5. The consequence is that schedulers using the reward policy will always *penalize* a process, whilst kernels using the penalty policy will *reward* processes. This is reflected in figs. 5.5b and 5.5c, as the look of the bars mimic the one of figure 5.5a.

Also the data collected with *wl2* confirm the experimental results discussed in section 5.3.1. For the runs using the L2 and L3 caches' PMCs it is clear how the *penalty* algorithms provide benefits and the schedulers using the *reward* system slow down the execution. The reward algorithm is able to provide a boost in performance only when considering the L1 cache PMCs and weighting more the cache miss rate, although it is limited when compared to the speed-up obtained by the penalty algorithms on the same cache level. The scaled branch miss rate plays an important role in differentiating the results between figs. 5.4a and 5.6a. For FFmpeg and Make it assumes the value of 52.5% and 38% respectively, raising significantly the penalty. A penalty increase translate in less context switches opportunities, the key to better performance in this analysis.

Although PMC-based scheduling does not always improve performance and is extremely workload-dependent, the data collected from the L1 runs in section 5.3.1 provide an interesting insight: it is possible that the CFS assigns the timeslice in a sub-optimal manner when put under stress. In both workloads, *penalty* is always in the range 40-45%, which halves the timeslice computed by `sched_slice()`. Even in front of a considerable increase in context switches, for every workload the tasks are capable to end their execution earlier. This suggests that in the original version of the kernel the timeslice calculation assigns too large timeslices, which hurt multitasking performance. On the other hand, `vruntime` does not need any alteration. When the penalty

is applied to `delta_fair`, entities are pushed further in the red-black tree by the same factor, resulting in the same execution timeline as in the original kernel.

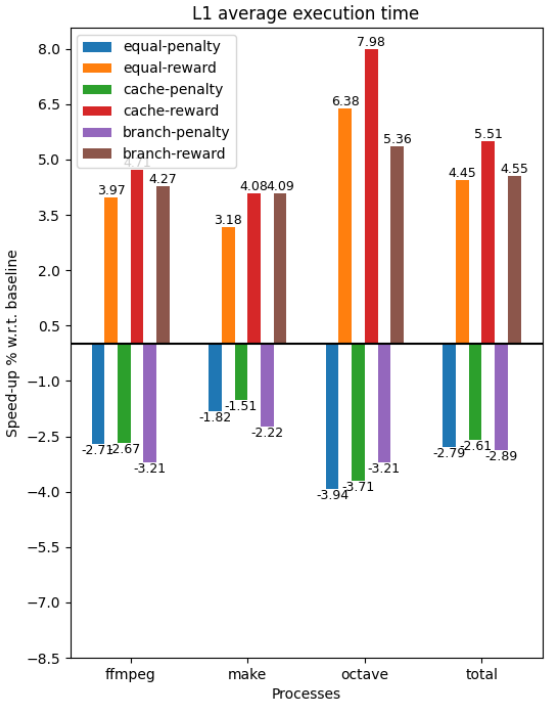
The data in this section supports this theory as well, even if they do so in a more limited way. Due to the interference of the branch miss rate, FFmpeg and Make penalties are not similar and differ significantly from Octave's ones, therefore the execution timeline differs between the modified and original kernels. But, even with a different execution order, both *wl1* and *wl2* receive a performance boost, confirming that an aggressive reduction of the timeslice raises the scheduler throughput.

		Original	EQ Penalty (%)	EQ Reward (%)	CA Penalty (%)	CA Reward (%)	BR Penalty (%)	BR Reward (%)
L1	CMR	6.097%	-2.396	-2.377	-0.926	-2.655	-1.654	-2.182
	BMR	25.494%	-1.220	-0.656	-1.463	0.376	0.312	-0.432
	Penalty		-34.540	34.274	-39.327	39.094	-29.471	29.426
	VCSW	46078	-0.126	0.360	0.091	0.172	0.353	0.370
	NVCSW	4652	-16.627	10.297	-16.772	10.805	-17.875	9.478
L2	CMR	48.156%	-0.692	0.262	-0.877	-0.464	-1.358	-0.334
	BMR	25.345%	-1.322	1.887	-1.974	1.530	-0.823	0.139
	Penalty		-13.706	12.875	-8.147	7.581	-19.428	18.927
	VCSW	46241	-0.531	0.030	-0.837	0.382	-0.267	-0.315
	NVCSW	4775	-6.707	-4.693	-1.666	-5.218	-10.934	3.101
L3	CMR	45.371%	-1.564	4.142	-2.047	1.696	-2.284	-0.350
	BMR	25.416%	0.276	0.361	-0.467	0.949	-1.017	-0.701
	Penalty		-15.038	13.537	-10.486	8.928	-20.197	19.724
	VCSW	46112	-0.616	0.426	-1.075	0.582	-0.503	0.220
	NVCSW	4781	-14.020	2.149	-8.922	4.409	-11.101	4.993

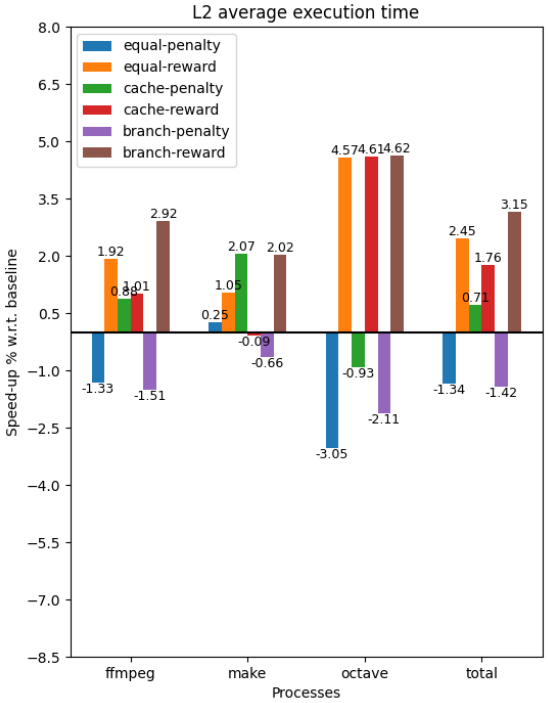
Table 5.7: ffmpeg wl1 cache branch data. CMR = cache miss rate, BRM = branch miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches

		Original	EQ Penalty (%)	EQ Reward (%)	CA Penalty (%)	CA Reward (%)	BR Penalty (%)	BR Reward (%)
L1	CMR	4.397%	2.027	-4.174	2.345	-4.497	0.945	-3.214
	BMR	35.388%	3.451	-4.070	2.996	-3.452	3.446	-4.414
	Penalty		-28.264	29.910	-35.960	37.122	-20.642	22.779
	VCSW	1252	-1.550	-0.024	-1.071	-1.398	-1.177	-0.051
	NVCSW	10139	-17.463	19.493	-21.214	24.428	-14.926	10.254
L2	CMR	42.753%	1.819	-2.788	2.423	-2.916	1.334	-2.900
	BMR	35.384%	2.272	-3.717	2.088	-2.443	2.835	-4.568
	Penalty		-10.001	11.912	-8.204	10.150	-11.611	13.918
	VCSW	1253	1.698	1.538	0.037	-1.203	-0.737	0.950
	NVCSW	10317	-13.126	5.755	-13.605	4.628	-12.836	7.490
L3	CMR	30.428%	-2.290	3.848	-0.637	1.806	0.816	-1.610
	BMR	35.348%	2.148	-2.450	1.152	-1.112	2.701	-4.125
	Penalty		-16.761	16.618	-18.199	17.791	-14.778	16.652
	VCSW	1268	-2.229	-2.713	-2.258	-3.415	-0.513	-1.241
	NVCSW	10496	-18.338	8.984	-17.672	9.355	-12.414	6.653

Table 5.8: make wl1 cache branch data. CMR = cache miss rate, BRM = branch miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches

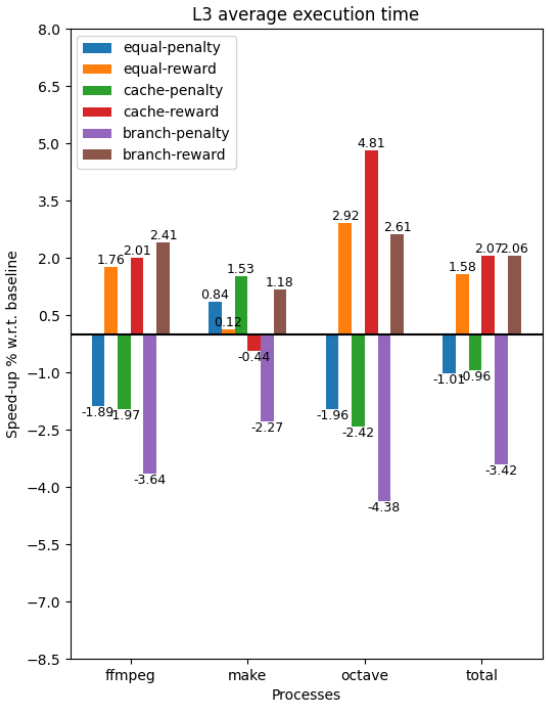


(a) L1



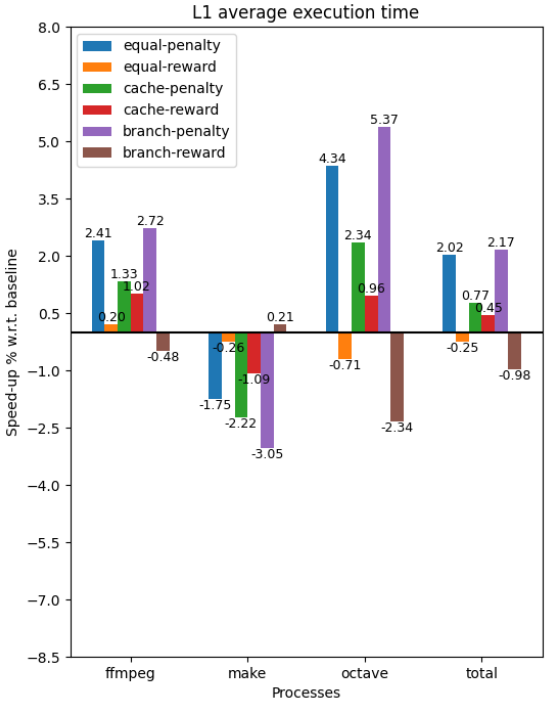
(b) L2

Figure 5.5: Cache PMCs wl1 results

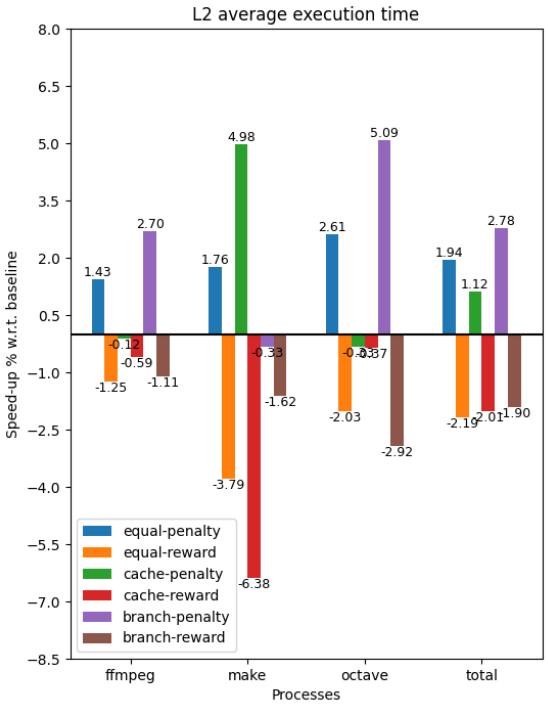


(c) L3

Figure 5.5: Cache PMCs wl1 results (cont.)

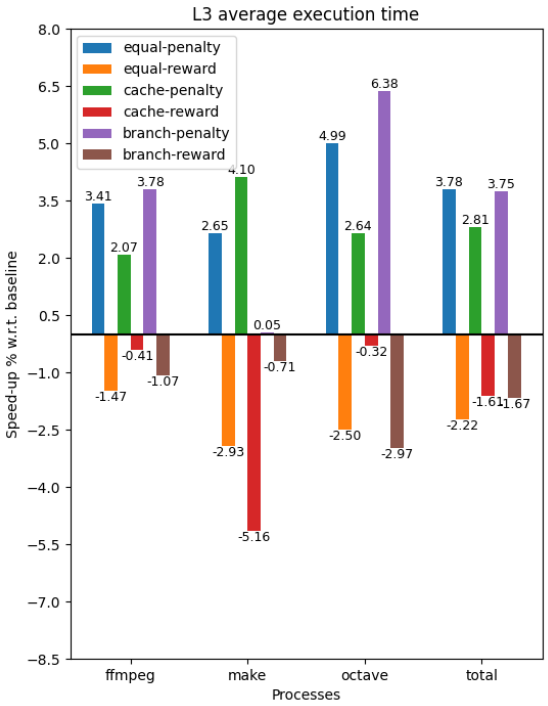


(a) L1



(b) L2

Figure 5.6: Cache PMCs wl2 results



(a) L3

Figure 5.7: Cache PMCs wl2 results (cont.)

		Original	EQ Penalty (%)	EQ Reward (%)	CA Penalty (%)	CA Reward (%)	BR Penalty (%)	BR Reward (%)
L1	CMR	2.286%	-14.733	-6.554	-8.477	-7.435	-11.581	5.205
	BMR	15.989%	-5.190	-7.107	-3.005	-6.694	-4.523	-4.009
	Penalty		-41.542	41.429	-44.706	44.653	-38.200	37.846
	VCSW	267	-20.477	25.983	-23.911	20.864	-23.311	22.400
	NVCSW	7764138	19.661	-25.414	24.995	-27.057	19.097	-18.082
L2	CMR	63.678%	-4.193	-0.590	-4.445	0.965	-10.886	2.199
	BMR	16.371%	-6.777	-8.918	-5.440	-11.282	-7.451	-6.241
	Penalty		-11.988	10.812	-0.657	-1.897	-24.603	22.187
	VCSW	222	13.006	5.056	26.403	8.956	-6.856	27.963
	NVCSW	8194734	3.920	-18.385	-2.681	-20.723	2.488	-18.421
L3	CMR	56.518%	-9.180	1.612	-7.260	5.564	-6.963	-2.150
	BMR	16.320%	-5.045	-7.647	-5.872	-6.653	-6.698	-8.649
	Penalty		-16.696	13.638	-6.983	1.362	-25.554	24.933
	VCSW	246	-16.970	19.669	-22.301	22.450	-21.432	17.634
	NVCSW	8183025	6.524	-18.105	2.255	-18.590	11.110	-21.592

Table 5.9: octave wl1 cache branch data. CMR = cache miss rate, BRM = branch miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches

		Original	EQ Penalty (%)	EQ Reward (%)	CA Penalty (%)	CA Reward (%)	BR Penalty (%)	BR Reward (%)
L1	CMR	3.745%	-1.281	-1.239	-3.047	-0.961	0.823	-1.493
	BMR	52.533%	0.293	-0.494	-0.574	-0.148	0.667	-1.128
	Penalty		-21.901	21.922	-34.330	34.033	-9.545	10.080
	VCSW	8292	-13.050	9.462	-12.135	5.269	-13.760	12.548
	NVCSW	15262	-23.162	20.648	-19.272	17.844	-22.613	21.968
L2	CMR	43.502%	-0.249	-0.493	0.065	-0.423	-1.145	0.267
	BMR	52.456%	0.710	-0.671	-0.288	0.267	0.871	-0.954
	Penalty		-2.021	2.240	-4.444	4.334	0.265	0.102
	VCSW	8368	-5.735	3.565	1.025	-1.911	-10.848	7.894
	NVCSW	15794	-8.726	8.236	-2.101	-1.176	-18.366	13.206
L3	CMR	28.682%	-8.155	-0.799	-5.886	0.686	-4.086	0.869
	BMR	52.551%	-0.034	0.132	-0.888	0.747	0.634	-0.632
	Penalty		-10.669	9.378	-16.865	15.045	-3.616	3.566
	VCSW	8352	-9.532	7.190	-6.297	0.828	-12.883	9.411
	NVCSW	15309	-15.648	12.054	-11.166	6.357	-21.512	18.137

Table 5.10: ffmpeg wl2 cache branch data. CMR = cache miss rate, BRM = branch miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches

		Original	EQ Penalty (%)	EQ Reward (%)	CA Penalty (%)	CA Reward (%)	BR Penalty (%)	BR Reward (%)
L1	CMR	5.285%	1.749	-5.617	2.201	-6.208	1.163	-4.279
	BMR	38.015%	4.382	-7.538	4.316	-6.985	3.722	-7.113
	Penalty		-25.152	28.165	-33.032	35.266	-17.519	20.937
	VCSW	10285	-0.479	0.232	-0.461	0.337	-0.427	0.174
	NVCSW	35749	-16.349	34.949	-17.227	37.683	-13.411	26.476
L2	CMR	45.988%	2.247	-1.360	2.144	-1.632	1.710	-1.642
	BMR	37.929%	2.024	-4.419	2.098	-3.474	3.380	-5.750
	Penalty		-6.899	8.833	-5.159	6.858	-8.342	11.299
	VCSW	10270	-0.124	0.328	-0.007	0.223	-0.141	0.240
	NVCSW	36855	-2.358	13.298	-6.725	10.492	-10.167	16.358
L3	CMR	33.685%	0.985	-5.685	2.172	-3.888	4.414	-4.827
	BMR	38.029%	3.192	-5.939	2.960	-4.747	3.228	-6.504
	Penalty		-12.702	15.625	-13.916	16.221	-11.045	14.611
	VCSW	10287	-0.478	0.207	-0.429	0.215	-0.351	0.114
	NVCSW	35933	-12.686	24.083	-12.810	20.680	-11.072	22.808

Table 5.11: make wl2 cache branch data. CMR = cache miss rate, BRM = branch miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches

		Origi-nal	EQ Penalty (%)	EQ Reward (%)	CA Penalty (%)	CA Reward (%)	BR Penalty (%)	BR Reward (%)
L1	CMR	1.788%	-16.133	-8.724	-7.008	-13.401	-9.825	3.101
	BMR	14.107%	-4.766	-8.627	-4.190	-8.554	-5.095	-8.412
	Penalty		-42.629	42.658	-45.526	45.531	-39.678	39.779
	VCSW	320	-1.238	-7.220	-15.668	2.341	-12.609	-4.869
	NVCSW	56019507	5.396	-12.937	8.969	-12.792	3.858	-11.359
L2	CMR	69.267%	-9.518	-4.638	-6.227	-6.029	-14.547	1.990
	BMR	14.033%	-4.705	-7.947	-6.322	-7.904	-5.033	-7.949
	Penalty		-12.095	10.440	1.857	-2.112	-25.356	22.583
	VCSW	311	-14.290	5.232	-1.780	8.308	-8.330	18.064
	NVCSW	54899482	3.854	-6.531	-0.108	-5.368	1.082	-7.724
L3	CMR	43.226%	-14.626	-1.607	-17.374	6.247	-11.153	-1.214
	BMR	14.168%	-5.783	-8.373	-6.273	-7.337	-5.592	-7.597
	Penalty		-24.924	22.167	-20.003	12.161	-30.471	29.395
	VCSW	336	-14.624	-16.617	-20.920	-9.340	-15.735	-2.042
	NVCSW	56423324	-1.245	-8.290	-1.900	-3.955	-0.182	-6.619

Table 5.12: octave wl2 cache branch data. CMR = cache miss rate, BRM = branch miss rate, VCSW = voluntary context switches, NVCSW = non-voluntary context switches

Chapter 6

Conclusions

This thesis introduced a penalty/reward system based on performance counters in the Linux Completely Fair Scheduler to alter scheduling decisions. To access these hardware facilities, part of this work dealt with the development of a custom PMC library providing almost no overhead. The scheduler relies on such framework to collect performance counter data during context switches to evaluate threads' behavior. In particular, the kernel is instructed to monitor the caches and the branch prediction units. Once the scheduler is in possess of the data, it processes it to compute a percentage value called *penalty*. It is a quantity comprised between -50% and 50%, representing the thread's global intensiveness on the monitored resources. A negative value means that the task shall be rewarded, a positive value that shall be penalized. To operate on number in $\mathbb{R}$ without exploiting the FPU, as per kernel rules, a kernel fixed point library has been developed as well.

Within CFS, *penalty* is applied to two different parameters: the *virtual runtime* and the *timeslice*. The former determines the tasks execution timeline, while the latter is the maximum amount of time a task can expend on the CPU before being preempted. Depending on *penalty*'s sign and value, these quantities are modified to change scheduling behavior. By increasing (decreasing) the virtual runtime of a task, its chances to be scheduled soon are reduced (augmented). At the same time, by increasing (decreasing) the timeslice assigned to a task, the amount of work executable in a scheduling period varies as well. By tuning these variables the resource utilization can be improved, leading to better performance.

To systematically explore the design space, some kernel configurations invert *penalty* before its use, to penalize threads using fewer resources and rewarding tasks putting them under stress. Furthermore, this work analyzes the case exploiting only caches' PMCs and the case gathering data from both cache and branch predictors' PMCs. For the latter, the raw PMCs metrics are combined with different weights to study the best configuration.

The kernel has been benchmarked using real-world applications, which are *FFmpeg* [1], *Make* [12] and *GNU Octave* [10]. These executables are capable of putting under

heavy stress the scheduler, the CPU and the memory subsystem thanks to their parallelism, and in this context are referred to as *heavyweight processes*. Additionally, another set of programs perform a perturbative action to increase the scheduler's stress. These are an embedded implementation of the AES-256 algorithm [36], an FFT algorithm [17] and a lightweight image recognition library [17]. They are called *lightweight processes*. The heavyweight processes execute two different kind of workload, labeled *wl1* and *wl2*, to analyze the scheduler's behavior under different working conditions. With respect to *wl1*, *wl2* is generally more resource-intensive and requires more time to complete.

A Python script launches in parallel a user-space logger, to collect the kernel's output, and all the heavyweight processes. Until they have not ended their execution, the script starts after a random amount of time one of the lightweight processes and wait for the end of its execution. The script repeats these steps 30 times for statistical consistency, then changes the workload and restarts.

For the cache PMCs-only scenario, results for *wl1* show how the kernels using the *reward* algorithm, that is the kernels rewarding tasks putting under heavy stress the caches, provide a consistent speed-up regardless of the cache level the PMCs are tied to. However, while for the L2 and L3 caches PMCs the reward algorithm is effectively rewarding cache-intensive tasks, with the L1 is penalizing every thread. *wl2* almost flip the outcome of *wl1*: in the L1 cache the reward algorithm is still providing better results, although more modest when compared to the previous workload. However, for the L2 and the L3 caches, the *penalty* algorithm proved to be more effective.

For *wl1*, adding the branch miss rate does not significantly change the results. The *reward* algorithm enhances CFS performance using the PMCs of any cache and regardless of the miss rates weights. The results confirm also *wl2*'s previous analysis, which favors the penalty algorithm for the L2 and L3 caches. With the L1 PMCs, though, both algorithms speed-up the execution of the processes when compared to the original kernel.

Results are consistent across all measurements only when the L1 PMCs are in use and the kernel is exploiting the *reward* algorithm. Although PMC-based scheduling is heavily workload-dependent, it indirectly suggests that CFS may assign the timeslice in a suboptimal manner when put under stress. With the *reward* system using the L1 PMCs, the timeslice assigned by the kernel is almost half the original one, and even in front of a significant increase of context switches and their overhead, the tasks' execution times improve significantly. On the other hand, due to the similar *penalty value*, especially in the cache PMCs only scenario, the execution timeline is the same as the original kernel and hence the *virtual runtime* does not affect the result.

Future works may study if the timeslice calculation may effectively be improved for the CFS. Moreover, the PMCs knowledge could be exploited in the load balancing algorithm to improve caches utilization. Furthermore, it would be interesting to study if the power efficiency could be improved by monitoring the tasks' power consumption.

Another line of work could explore if this work performs better in other scheduling algorithms or in other systems like server or embedded platforms, using an alternative

architecture to x86-64.

Bibliography

[1] Fabrice Bellard, Bobby Bingham, and FFmpeg Team. *FFmpeg*. 2000-2021. URL: https://ffmpeg.org/ (visited on 10/29/2021).

[2] Irene Finocchi Camil Demetrescu and Giuseppe F. Italiano. *Algoritmi e strutture dati*. Italian. Ed. by Paolo Roncoroni. 2nd. McGraw-Hill, 2008.

[3] Po-Yung Chang et al. "Branch Classification: A New Mechanism for Improving Branch Predictor Performance." In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 22–31. ISBN: 0897917073. DOI: 10.1145/192724.192727. URL: https://doi.org/10.1145/192724.192727.

[4] *clone(2) Linux User's Manual*. Mar. 2021.

[5] TIS Committee. *Tool Interface Standard (TIS). Executable and Linking Format (ELF)*. Version 1.2. May 1995.

[6] *Control Groups*. The Linux kernel community. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html (visited on 09/18/2021).

[7] corbet. *How fast should HZ be?* URL: https://lwn.net/Articles/145973/ (visited on 10/14/2021).

[8] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel*. 3rd ed. O'Really, Nov. 2005.

[9] *Debugging the kernel using Ftrace - part 1*. URL: https://lwn.net/Articles/365835/ (visited on 10/26/2021).

[10] John W. Eaton, James B. Rawlings, John G. Ekerdt, et al. *GNU Octave*. 1998-2021. URL: https://www.gnu.org/software/octave/ (visited on 10/29/2021).

[11] Wei-Cong Fan et al. "Comparison of Interactivity Performance of Linux CFS and Windows 10 CPU Schedulers." In: *2020 International Conference on Green and Human Information Technology (ICGHIT)*. 2020, pp. 31–34. DOI: 10.1109/ICGHIT49656.2020.00014.

[12] Stuard Feldman et al. *Make*. 1976-2021. URL: https://www.gnu.org/software/make/ (visited on 10/29/2021).

[13] Bryan Ford and Sai Susarla. "CPU Inheritance Scheduling." In: *SIGOPS Oper. Syst. Rev.* 30.SI (Oct. 1996), pp. 91–105. ISSN: 0163-5980. DOI: 10.1145/248155.238765. URL: https://doi.org/10.1145/248155.238765.

[14] *ftrace - Function tracer*. URL: https://www.kernel.org/doc/html/latest/trace/ftrace.html (visited on 10/26/2021).

[15] *getpid(2) Linux User's Manual*. Mar. 2021.

[16] Thomas Gleixner and Douglas Niehaus. "Hrtimers and Beyond: Transforming the Linux Time Subsystems." In: *Proceedings of the Ottawa Linux Symposium*. 2006.

[17] M.R. Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite." In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[18] *HOWTO do Linux kernel development*. (Visited on 10/28/2021).

[19] Wei-Lien Hsu. "Video transcoding using GPU accelerated decoder." In: *Parallel Processing for Imaging Applications*. Ed. by John D. Owens et al. Vol. 7872. International Society for Optics and Photonics. SPIE, 2011, pp. 147–155. DOI: 10.1117/12.876569. URL: https://doi.org/10.1117/12.876569.

[20] D. Hunt. "Advanced performance features of the 64-bit PA-8000." In: *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway*. 1995, pp. 123–128. DOI: 10.1109/CMPCON.1995.512374.

[21] *Intel Performance Monitorin Units Reference*. Intel Corporation. URL: https://perfmon-events.intel.com/ (visited on 09/09/2021).

[22] *Intel® 64 and IA-32 Architectures Software Developer Manual. Volume 4*. Intel Corporation. 2020.

[23] *Intel® 64 and IA-32 Architectures Software Developer Manual. Volume 3 (3A, 3B, 3C & 3D)*. Intel Corporation. 2020.

[24] Ciji Isen, Lizy K. John, and Eugene John. "A Tale of Two Processors: Revisiting the RISC-CISC Debate." In: *Computer Performance Evaluation and Benchmarking*. Ed. by David Kaeli and Kai Sachs. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 57–76. ISBN: 978-3-540-93799-9.

[25] D. A. Jimenez and C. Lin. "Dynamic branch prediction with perceptrons." In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 2001, pp. 197–206. DOI: 10.1109/HPCA.2001.903263.

[26] *Kernel Stacks*. The Linux kernel community. URL: https://www.kernel.org/doc/html/latest/x86/kernel-stacks.html (visited on 09/17/2021).

[27] Joonwon Lee and Umakishore Ramachandran. "Synchronization with Multiprocessor Caches." In: *SIGARCH Comput. Archit. News* 18.2SI (May 1990), pp. 27–37. ISSN: 0163-5964. DOI: 10.1145/325096.325107. URL: https://doi.org/10.1145/325096.325107.

[28] Juri Lelli et al. "An experimental comparison of different real-time schedulers on multicore systems." In: *Journal of Systems and Software* 85.10 (2012). Automated Software Evolution, pp. 2405–2416. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.05.048. URL: https://www.sciencedirect.com/science/article/pii/S016412121200146X.

[29] *Linux CFS*. The Linux kernel community. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (visited on 10/03/2021).

[30] Robert Love. *Linux Kernel Development*. 3rd ed. Pearson Education, 2010.

[31] Marshall Kirk Mckusick and Michael J. Karels. "A New Virtual Memory Implementation for Berkeley UNIX." In: *EUUG Conference Proceedings (Autumn*. 1986, pp. 451–458.

[32] *Message logging with printk*. URL: https://www.kernel.org/doc/html/latest/core-api/printk-basics.html (visited on 10/26/2021).

[33] *Mobile 4th Generation Intel® Core™ Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family. Specification Update*. Version 038US. Intel Corporation. Apr. 2020.

[34] Ingo Molnar. *[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]*. URL: https://lwn.net/Articles/230501/ (visited on 10/04/2021).

[35] *perf: Linux profiling with performance counters*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 11/08/2021).

[36] Blu5 View Pte et al. *SEcube SDK*. 2016-2021. URL: https://github.com/SEcube-Project/SEcube-SDK (visited on 10/29/2021).

[37] E. Rotenberg, S. Bennett, and J.E. Smith. "Trace cache: a low latency approach to high bandwidth instruction fetching." In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. 1996, pp. 24–34. DOI: 10.1109/MICRO.1996.566447.

[38] J. C. Saez et al. "PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler." In: *The Computer Journal* 60.1 (Jan. 2017), pp. 60–85. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxw065. eprint: https://academic.oup.com/comjnl/article-pdf/60/1/60/10329287/bxw065.pdf. URL: https://doi.org/10.1093/comjnl/bxw065.

[39] *Sandy Bridge (client)*. Wikichip. URL: https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client) (visited on 11/05/2021).

[40] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 10th ed. John Wiley & Sons, Inc., 2018.

[41] Dan Terpstra et al. "Collecting Performance Data with PAPI-C." In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173. ISBN: 978-3-642-11261-4.

[42] Paul Turner, Bharata B Rao, and Nikhil Rao. "CPU bandwidth control for CFS." In: *Proceedings of the Linux Symposium*. 2010, pp. 245–254. URL: http://www.linuxsymposium.org/LS_2010_Proceedings_Draft.pdf.

[43] Srivatsa Vaddagiri. *Add group awareness to CFS*. URL: https://lwn.net/Articles/239619/ (visited on 10/05/2021).

[44] Vincent M. Weaver. "Self-monitoring overhead of the Linux perf_event performance counter interface." In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 102–111. DOI: 10.1109/ISPASS.2015.7095789.

[45] A. W. Wilson. "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors." In: *Proceedings of the 14th Annual International Symposium on Computer Architecture*. ISCA '87. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1987, pp. 244–252. ISBN: 0818607769. DOI: 10.1145/30350.30378. URL: https://doi.org/10.1145/30350.30378.

[46] C.S. Wong et al. "Fairness and interactive performance of O(1) and CFS Linux kernel schedulers." In: *2008 International Symposium on Information Technology*. Vol. 4. 2008, pp. 1–8. DOI: 10.1109/ITSIM.2008.4631872.

[47] Tse-Yu Yeh and Yale N. Patt. "Two-Level Adaptive Training Branch Prediction." In: *Proceedings of the 24th Annual International Symposium on Microarchitecture*. MICRO 24. Albuquerque, New Mexico, Puerto Rico: Association for Computing Machinery, 1991, pp. 51–61. ISBN: 0897914600. DOI: 10.1145/123465.123475. URL: https://doi.org/10.1145/123465.123475.

[48] Yong Zhao et al. "Preemptive Multi-Queue Fair Queuing." In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 147–158. ISBN: 9781450366700. DOI: 10.1145/3307681.3326605. URL: https://doi.org/10.1145/3307681.3326605.

[49] Peter Zijlstra. *sched: high-res preemption tick*. URL: https://lwn.net/Articles/230501/ (visited on 10/14/2021).