POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

MSC DEGREE IN MECHATRONIC ENGINEERING

MSC DEGREE THESIS

New Scheduling Approaches for Linux OS

Supervisors:
Prof. Maurizio REBAUDENGO
Researcher Alessandro SAVINO

Candidate:
Graziano Mario FANIZZI

DECEMBER 2019

Declaration of Authorship

I, Graziano Mario FANIZZI, declare that this thesis titled, “New Scheduling Approaches for Linux OS” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a MSc degree at Politecnico di Torino.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Politecnico di Torino or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

“Intelligence is the ability to avoid doing work, yet getting the work done.”

Linus Torvalds

POLITECNICO DI TORINO

Abstract

Mechatronic Engineering
Department of Control and Computer Engineering

Master's Degree

New Scheduling Approaches for Linux OS

by Graziano Mario FANIZZI

Performance counters have been proven effective in characterising applications and system performances. Despite several examples of their usage in this regard already exist at user level, their potential and application to support scheduling decisions has not been fully explored yet.

The goal of this thesis is indeed to enable the Linux kernel to observe and monitor the Last-Level Cache behaviour by accessing performance counters data, in particular the cache-miss rate for each task. These observations are then used to modify the current Linux Completely Fair Scheduler, the default scheduler for standard processes, and eventually characterise the behaviour of the proposed patch by testing it on a multithreading test environment and observing the related experimental results.

Contents

List of Figures

List of Tables

Introduction

The increasing complexity of real world applications, in both scientific and technical fields, has recently posed new challenges in terms of computational power and capabilities provided by nowadays computer systems. In this regard, microprocessors, which all modern computers are equipped with, are asked to provide them with higher and higher performances, in order to satisfy the demand coming from the most ambitious scientific questions as well as the cutting edge of modern technology and engineering.

Microprocessors constitute the basic hardware chips which most Central Processing Units (CPUs) are based on. They can be found on all modern computer systems, stretching from desktop machines to large servers or even embedded systems, depending on the specific field of application. In order to keep up with this continuous demand of productivity and make the most of computer systems, multicore processors have now become universal. The number of cores of these devices has lately experienced an exponential growth, while the increase in core's frequency has been relatively small[23].

As a consequence, exploiting parallelism is currently the new trend to achieve such performances, leading to the development of complex and advanced multithreaded applications. In this sense, the operating system is responsible of handling all available resources and make them forthcoming to these user space programs, such that the overall performances of applications are not degraded. The availability of more processors introduces in general more issues for the OS, such as load-balancing and cache locality [23]; for machines provided with multicore chips, further complications arise from resources contention and sharing, forcing the OS to provide some effective synchronisation mechanism [11].

The OS scheduler plays a particularly critical role in terms of performances in multicore systems. Its functionality primarily entails the choice on which task has to run at any given time on each available core, as well as how long it ought to run before preempting it, taking into account different tasks priorities, too. In doing so, the scheduler shall also perform a load balancing action among the cores [16] while, at the same time, trying to exploit cache-locality, thus avoiding to migrate a task from one core to another, if useful runtime data are available in one of the cache-levels.

This work focuses on the default Linux scheduler, whose policy is employed to handle standard processes: the Completely Fair Scheduler (CFS) [20]. The main objective is to instrument the scheduler to retrieve per-task Performance Counters (PMCs) data and ultimately use them to alter the behaviour of the scheduling algorithm. These data concern the Last-Level Cache (LLC) monitoring by retrieving the number of cache-misses from special HW units called *Performance Monitoring Units*, that can be found on most microprocessors from the majority of microchip producers, such as Intel, ARM, AMD and so on and so forth.

These units allow to monitor system and applications performances by counting occurrences of specific events, which they can be programmed to count[8]. While historically PMCs have been widely used in Linux to monitor user-space applications performances [9, 24], their employment in the scheduling area has been moderately explored [15, 21, 6, 17, 10, 22, 2]. As a consequence, many SW libraries are available to access them and gather their data from user land programs. These tools can be essentially categorised in two families. The first embraces those utilities providing the user with access to PMCs by means of a set of command-line tools; in this category fall down SW packages like *OProfile* [7], *perfmon2* [13] and the more comprehensive *perf* [26]. Their functioning relies on creating external tasks that are responsible of retrieving PMCs information for one or more target user processes. The second family, instead, encompasses SW libraries that are used to access the counters from inside the target application source code. This is done by inserting proper function calls to the library, thus circumscribing the collection of PMCs values to a specific snippet of code; this methodology is implemented in tools such as *PAPI* [24, 5] and *libpfm*[13].

However, far fewer tools exist and can be effectively used to interface with PMCs at kernel level. These tools are essentially the `perf_event` Linux subsystem, on top of which *perf* is built but whose usage is often poorly and sparsely documented [25], and the open source tool PMCTrack [21, 14]. In this thesis, PMCs have been configured to count the number of LLC-misses and core cycles for each thread, aimed at pulling out a per-task cache-miss rate metric.

The first two chapters of this thesis constitute the background and the framework of this work. An overview of the scheduling in Linux systems, with a focus on the CFS scheduler, is provided in the first chapter. Instead, the second unit deals with Performance Counters and their functioning, with more emphasis on Intel's Performance Monitoring Units, since they constitute the available HW used for this project.

The third chapter is the bulk of this work. The scheduler's instrumentation to access PMCs will be described from a SW development point of view. Specifically, a PMCTrack-compatible kernel module [21, 14], developed to monitor the counters and retrieve per-task metrics, is first presented; eventually, the main CFS modifications and changes, relying on the data gathered by the above module, are then introduced.

In the fourth chapter, a testing environment for the modified version of CFS will be discussed. In particular, a subset of programs from a commercially representative benchmark [12] has been employed to design two test variants, in order to test and stress the new scheduler, aimed at comparing its behaviour with the default algorithm running on most Linux distributions. Furthermore, after going through the test environment and experimental setup, the results of both tests are graphically presented.

In the final conclusions, the results of the experiments conducted, as well as possible future works, are finally discussed.

Chapter 1

Linux process scheduling

1.1 Linux process management

1.1.1 Linux processes basics

The basic definition of a process, regardless the several modern operating systems running on everyday's computer system, is an instance of a program in execution [1].

The program is essentially an executable object code residing in one of the file systems. The executing program code, often called "text section" in Unix systems, is still a passive entity; in order to become a process, which is by definition an active entity, it needs to be read and loaded into memory.

The OS, from its perspective, needs to store and manage some information associated to the executable program to be used during its lifetime and allowing it to interact with the rest of the system and accomplish its goals. These additional resources are, among others: address space, open files for read/write operations, pending signals, internal kernel data, process state, one or more threads of execution and a data section, usually named as stack.

A process begins its life when it is created from its parent by means of a system call (syscall), and it is therefore called its child. When it is created, the child is almost an exact copy of its parent; in fact, it obtains the same address space and it executes the same source code of the parent, starting from the next instruction following the system of call that created it.

It is important to point out that, even if the child is an exact copy of the parent when it is created, it has a separate copy of data, i.e. stack and heap. This means that any change made by the child to a memory location are not visible in the parent process.

In Linux, however, an abstraction of the process is incomplete without introducing the concept of *threads*: threads of execution, usually called simply threads, are the smallest units of processing that the CPU can actually perform. Therefore, a process contains at least one thread of execution; if it contains multiple ones, it will be called a multi-threaded process.

The practical difference between processes and threads is that the former do not share the same address space among them while the latter do. Therefore, in order to let two processes share resources, some inter-process communication techniques, such as pipes, sockets and shared memory, are needed.

Previous versions of the Kernel did not support multi-threaded applications; therefore they were seen by the OS as normal processes. However, in earlier versions of Linux a new approach has been adopted, quite unique with respect to other operating systems, in order to allow the system to deal with multithreaded applications: the concept of *lightweight processes* [18, 4]. These are actually normal processes

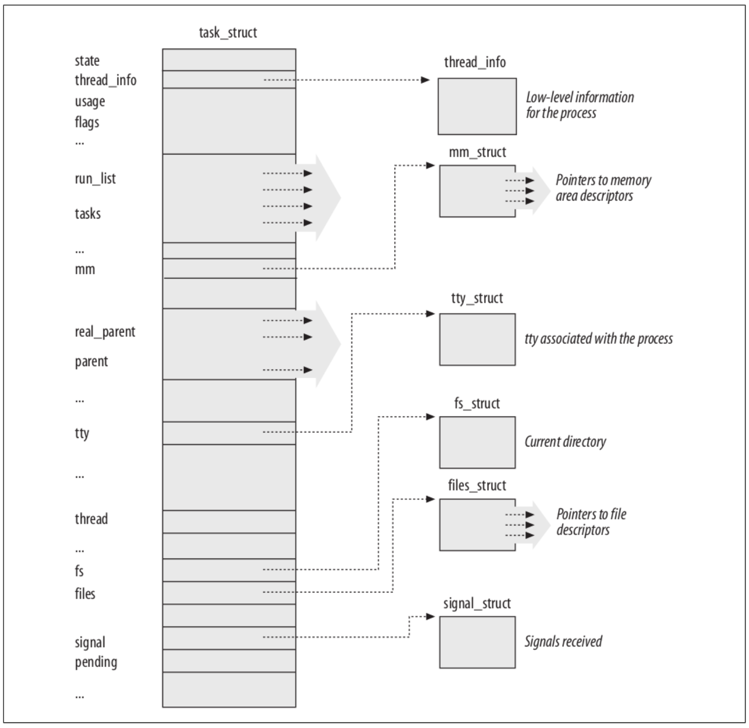


FIGURE 1.1: Linux process descriptor.

with the only difference that, provided an effective synchronisation method, they are capable of sharing some resources like memory address or files.

Once this distinction is made clear, it is therefore possible to state that, from the scheduling standpoint, in Linux there is no separate distinction between processes and threads, since they are both seen as ordinary processes by the kernel. As a consequence of this duality, most of the time both processes and threads are simply referenced in the kernel source code as *tasks*.

1.1.2 Process descriptor

The kernel keeps stored a list of all the processes, i.e. the *task list*, in a circular doubly linked list. Each entry of the task list includes all the information needed by the OS about each process and it is therefore named as *process descriptor*: it contains relevant information as process ID, state, relationships with other processes, e.g. parent and children, address space, processor registers, etc.

This structure is implemented as a structure of type `struct task_struct`, defined in the source file <linux/sched.h>. A schematic description highlighting the most important fields is shown in figure 1.1.

Doubly linked lists and task list

Before starting the analysis of the most relevant information contained in the process descriptor and how this is handled by the kernel, it is worth spending some words on doubly linked lists and their implementation in the kernel source code. This is important since, as already mentioned, the task list belongs to this data structure.

Usually a doubly linked list is a data structure like the following:

```
struct list_element {
  void *data;
```

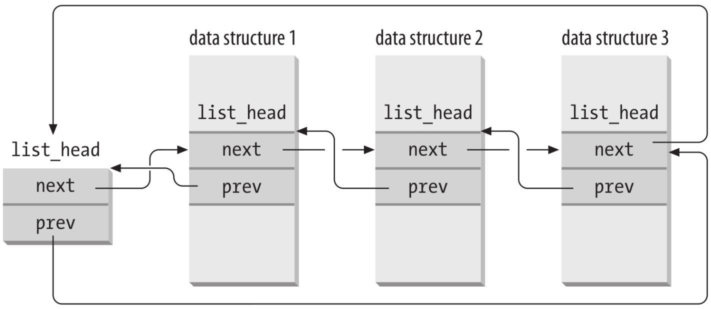


FIGURE 1.2: A doubly linked list.

```
  struct list_element *next; //next element
  struct list_element *prev; //previous element
};
```

where, regardless of the data, the two pointers `next` and `prev` point to the next and previous nodes of the list itself, respectively.

In Linux things are slightly different: instead of keeping a linked list structure, the approach is to embed a linked list node containing only forward and backward pointers to another node into the main data structure. A graphical representation is provided in figure 1.2.

This new structure is named `list_head` and it is defined in <include/linux/types.h>:

```
struct list_head {
struct list_head *next, *prev;
};
```

This new construct is meant to be placed as a member of a list element structure and to represent the list of all the nodes in the linked list:

```
struct list_element {
  void *data;
  struct list_head myList;  /*list of all the elements*/
};
```

This is what happens, in fact, in the task list itself: the fields of the structure `task_struct` that implement the links of the current process descriptor to the list of its children and siblings are exactly of type `list_head`. (see section Relationships among processes).

Some functions and macros are available in the kernel in order to manipulate linked list. Some of the most common and important include:

- `LIST_HEAD(list_name)` : create a new list
- `list_add(n, p)` : the element `n` pointed to by `p` is put immediately after the one pointed to by `p`
- `list_add_tail(n, p)` : the element `n` pointed to by `p` is put immediately before the one pointed to by `p`
- `list_entry(p, t, m)` : returns a pointer to the data structure of type `t` in which the `list_head` member with name `m` and address `p` is contained

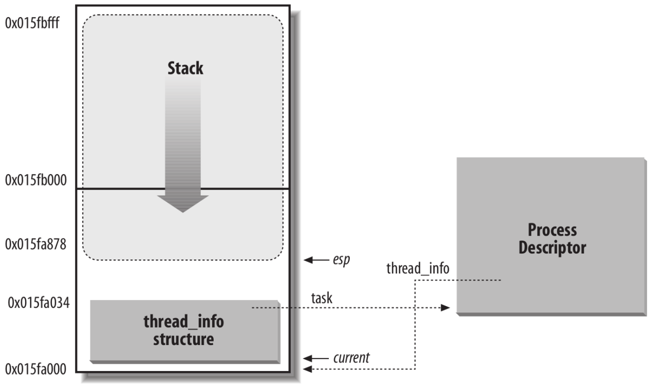


FIGURE 1.3: Process kernel stack and `thread_info` structure stored in two page frames

- `list_for_each(p, h)` : goes through the elements of the list whose head is `h`; at each iteration, `p` returns a pointer to the `list_head` structure for each element

Allocation

The `task_struct` is quite large in terms of size; furthermore, since it contains all the data needed by the OS in order to handle each process, it needs to be stored in dynamic memory.

In previous kernel versions, prior to 2.6, it was entirely placed at the end of the kernel stack of each process. In this way, it was possible to calculate its location directly by means of the stack pointer, which allows to address the stack's top location, without any extra register [4]; however, due to limitation in size of the kernel stack per process (usually around 8 kB, in two successive page frames) this was not satisfactory in terms of resource usage.

This consideration, together with the introduction of the SLAB allocator which dynamically creates the process descriptor [4], led to the implementation of another smaller structure, containing a pointer to the actual process descriptor: the `struct thread_info`.

For x86 architectures, it is defined in `arch/x86/include/asm/thread_info.h`:

```
struct thread_info {
struct task_struct  *task;     /* main task structure */
__u32       flags;     /* low level flags */
__u32       status;    /* thread synchronous flags */
__u32       cpu;     /* current CPU */
mm_segment_t      addr_limit;
unsigned int      sig_on_uaccess_error:1;
unsigned int      uaccess_err:1;   /* uaccess failed */
};
```

This new data structure is strongly dependent on the hardware architecture and it is still placed at the end of the kernel stack of the related process, i.e. bottom for stacks that grow down, up for stacks that grow up.

In figure 1.3, a memory portion of two pages containing the kernel stack and the `thread_info` structure is shown: the stack grows down, its end is referenced by the stack pointer `esp` and the linkage between the process descriptor and the `thread_info` structure is emphasised.

The esp register is used to access the location on top of the stack; with regards of x86 architectures, this starts at the end of the memory area and dynamically grows down. When a process moves from User to Kernel space, typically after the execution of a syscall, this stack is empty, thus having the stack pointer pointing to the first memory location, immediately after the kernel stack.

When data are put into the kernel stack, this chunk of memory starts growing while the value of esp is decreased accordingly.

In conclusion, the thread_info provides a fast and efficient way to access the task_struct of a process since:

- Its location is retrieved by simply reading the value of the esp register; once this address is obtained, it is possible to get the location of the structure task_struct directly from it
- Its size is quite small, around 52 bytes, leaving more free space to the kernel stack and allowing it to grow more

Process state

One of the most important field of the task_struct is definitely the state field. In Linux this field can assume the following different values:

- TASK_RUNNING : the process is runnable, i.e. either it is already running or it is located in a run-queue, waiting to be scheduled for running
- TASK_INTERRUPTIBLE : the process is blocked, based on a certain condition that has not happened yet. When the condition occurs, or if the process receives a signal, it is waken up and enters in TASK_RUNNING
- TASK_UNINTERRUPTIBLE : identical to TASK_INTERRUPTIBLE except that the process is not waken up by a signal
- TASK_STOPPED : the process execution has been paused after receiving a SIGSTOP, SIGTTIN or SIGTTOU signal
- TASK_TRACED : the process execution is monitored by another process, e.g. it has been stopped by a debugger program

Quite often the kernel needs to modify the state of a process, e.g. when it has to be stopped because it needs a not available resource or due to a context switch, and this is done by means of the set_task_state macro:

```
set_task_state(task, state);   /* set task task to state state
   */
```

In figure 1.4, a graphical scheme of the different states with the relative transitions is provided.

The above routine, apart from setting the new state, also ensures a safe operation, preventing the new assignment from being mixed with other instructions or reordered by the compiler or the CPU control unit. Furthermore, the kernel provides also the macro set_current_state(state), which is perfectly equivalent to set_task_state(state).

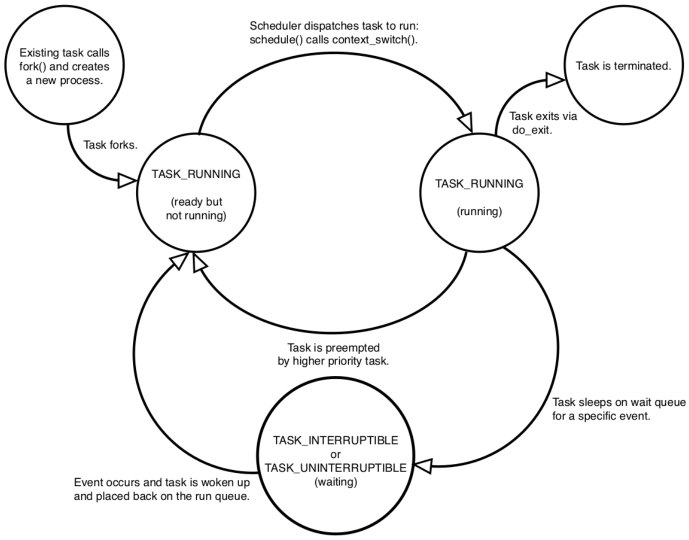

Existing task calls fork() and creates a new process.
Task forks.
Scheduler dispatches task to run: schedule() calls context_switch().
Task is terminated.
Task exits via do_exit.
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

FIGURE 1.4: Different states of a process and state transitions.

Process identifier

The OS keeps track of all the processes by assigning a unique number to each one: this is the *process identifier PID*. It goes without saying that this value needs to be accessed frequently by the kernel and it is therefore stored in the process descriptor, in the `pid` field.

Its opaque data type, i.e. whose physical representation is negligible, is of `pid_t`, usually an `int`; however, the default maximum value is 32768, which is the size of a `short int`. This is essentially due to backward compatibility with older kernel versions.

The maximum value of the PID is a relevant information for the system, since it defines an upper-bound on the number of processes that are allowed to exist concurrently. Furthermore, the PID values are allocated sequentially, providing an useful information about the creation order of the related processes; this useful notion gets lost as soon as the number of existing processes in the systems exceeds the maximum value of PID, and this can easily happen on large servers.

This is why Linux offers to the system administrator the capability to increase the default maximum, by simply editing the file `/proc/sys/kernel/pid_max`.

Referencing the current process

Typically, in most of the kernel source code, processes are directly referenced with a pointer to their `task_struct`; in order to enable this mechanism, allowing kernel developers to quickly access the process descriptor of the currently running task at any given time, the `current` macro comes into play.

This macro is obviously architecture-dependent: some architectures stores a pointer to the structure `task_struct` of the currently executing process in a register. In x86 architectures, as previously explained, the location of the structure `task_struct` is obtained by first getting the address of the `thread_info` structure, residing at the end of the kernel stack, which is in turn retrieved by the CPU stack pointer.

More specifically, for this kind of architecture, `current` makes use of `current_thread_info()` to retrieve the `thread_info` structure. Then, the `task` field of `thread_info` is dereferenced in order to obtain the structure `task_struct` of the process currently in execution.

Finally, it is worth clarifying that the `current` routine is available only when the kernel is in *process context*. Recalling the basic definition of process as a program in execution, this implies that its program code, embedded in an executable file, is executing and, therefore, this has to occur in some address space.

Usually a program starts executing in the *user-space*. However, when it needs some service provided by the OS, or when it throws an exception, the execution moves into the *kernel-space*. These two operations are performed by means of syscalls and exception handlers, respectively, which act as safe and pre-defined interfaces between the application and the kernel itself.

In these scenarios, the kernel is said to be "executing on behalf of the process", meaning that it is executing code that is related and tied to that specific task, i.e. in *process context*. Eventually, when the kernel has serviced the program request, either the process resumes its execution in user-space or in the meantime an higher-priority task has entered the `RUNNABLE` state, thus the scheduler needs to be invoked to select it for execution.

Relationships among processes

In Linux, as well as in general Unix systems, all the processes in the system belong to the same hierarchic family tree. In particular, each process has a unique *parent* and one or more *children*. Direct children of a same parent process are named *siblings*.

Going through these parent/child relationships in a bottom-up fashion leads to one special process, which all the others descend from: the `init` process. This is the very first process that is created in the system, with unitary PID value, and it is started by the kernel as the last step of the *boot loader*. It is in charge of reading the *initscripts* and actually starting the whole operating system.

The information about the direct relationships of each process is also provided in the process descriptor. In particular, for a generic process P, the members of the structure `task_struct` of P that exploit these relationships are:

- `real_parent`: either a pointer to the process descriptor of P's parent or, if the parent is dead, to *init*
- `parent`: pointer to the process descriptor of the parent of P
- `children`: head of the list of children of P
- `sibling`: link to the list of siblings of P

In practical terms, thanks to the above fields in the process descriptor, it is straightforward to obtain the process descriptor of the parent process, given the current one:

```c
struct task_struct *my_parent = current->parent;
```

Using the `list_for_each` and `list_entry` routines, it is possible to iterate over the list of children for the current process:

```c
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children) {
  task = list_entry(list, struct task_struct, sibling);
}
```

Using `list_entry` allows to easily obtain the next and previous elements of the task list, starting from a given one:

```
list_entry(task->tasks.next, struct task_struct, tasks)
list_entry(task->tasks.prev, struct task_struct, tasks)
```

These two routines can be invoked directly by means of the two macros `next_task` and `prev_task`.

Finally, it is possible to iterate over the entire task list by calling `for_each_process`:

```
  for_each_process(task) {
  printk("PID : %d\n", task->pid);  /*print the PID of each
   task*/
}
```

At each iteration `task` points to the process descriptor of the next task in the list and the value of PID is printed out.

1.1.3 Process creation

As mentioned at the beginning of the chapter, a child process begins its life when it is created by its parent. This operation is performed by means of a special and fundamental syscall: the function `fork()`.

Ii is worth pointing out that, in other operating systems, a unique syscall is adopted in order to create a new process into a new address space, load an executable file and start executing it. Conversely, in Linux, the `fork()` syscall is only in charge of creating a new child process which is, in the first place, an exact copy of its parent, i.e. the calling task; at this point, it is possible to either load an executable into the child's address space by calling a different syscall, i.e. `exec()`, or executing other custom code, different from the parent one.

In the latter case, in order to differentiate the sections of code that shall be executed from the child and from the parent, on success `fork()` returns the PID of the child process in the parent while 0 is returned in the child.

The new *forked* child is identical to the parent, except for its new PID in order to make it recognisable in the system, its PPID, i.e. its parent's PID that is set to the PID of the process calling `fork()` and few other resources and statistics.

Copy-on-write

One observation that might raise up regarding the `fork()` approach is certainly about efficiency. If each time a new child process is generated the entire address space of its parent is duplicated, then this turns out to be a waste of resources whenever these resources are not actually used, e.g. the child issues an `exec()` function which destroys the entire address space just copied.

In order to prevent the system from such ugly and inefficient scenarios, the `fork()` function is implemented using a *copy-on-write* approach. Basically, the idea is to avoid copying the data in case they are not going to be used: the actual duplication of resources is triggered only when the child attempts to use them for write operations; up to this point, instead, they are shared for read-only.

In practical terms, each page of the parent's address space is not copied into the child's address space until a child process writes to it; when this happens, the kernel copies its contents into a new page which is given to the child, thus allowing it to safely write on it. The limit case is when no page is ever written, implying that no page is ever copied, which is indeed what happens when the child process calls `exec()` immediately after `fork()`.

Fork details

The implementation of `fork()` is essentially a wrapper around the super syscall `clone()`. A set of flags is passed to `clone()` in order to specify which resources will be shared between child and parent, if any.

The possibility to specify these flags, effectively enabling two processes to share resources, leads us back to the conceptual difference between processes and threads. As stated at the beginning of this chapter, the kernel does not make any distinction between processes and threads. The reason relies exactly on the fact that they are both created by means of the very same syscall `clone()`, simply by passing to it different flags in the two cases.

Specifically, in order to create a thread, the `clone()` syscall is invoked with the following arguments:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

where the first argument is a logical sum of flags, allowing the child process to share, in order, address space, filesystem resources, file descriptors and signal handlers.

Conversely, it is also possible to *fork* a standard process by setting to zero the above flags:

```
clone(SIGCHLD, 0);
```

As a result of this approach, there is no need, from the kernel perspective, to provide any special data structure or support to handle, and also to schedule, threads, which are *de facto* simply processes that can share resources with other processes.

Subsequently the `clone()` syscall invokes the function `do_fork()`, whose definition is provided in <kernel/fork.c>. The purposes of this function are to copy the parent's process and start running the newly created child.

Copying the process is performed by calling `copy_process()`. The work done by this other function can be summed up as follows:

- Use `dup_task_struct()` to duplicate the parent's process descriptor , i.e. create new kernel stack and two new `thread_info` and `task_struct` structures
- Check if the new child creation is feasible in terms of system resources and number of processes assigned to the current user
- Clear and/or initialise some fields containing statistical information in the process descriptor
- Set child's state to `TASK_UNINTERRUPTIBLE`
- Call `copy_flags()` to update the `flags` field of the process descriptor; in particular, `PF_SUPERPRIV` is cleared, indicating that the process has not used its super-user privileges and `PF_FORKNOEXEC` is set, denoting that the child has not issued an`exec()` syscall, yet
- `alloc_pid()` provides the child process with a new PID value
- Depending on the input flags passed to `clone()`, the resources selected to be shared are then copied
- Return a pointer to the new child

1.1.4 Process destruction

Just like it was born, every process must also die. This happens, in the brightest of the scenarios, when its program has accomplished the goals it was written for. In this case, either the program invokes its own termination by calling the `exit()` syscall or the compiler implicitly executes it when the `main()` program routine returns successfully.

It might also happen, in less optimistic situations, that the process receives a signal that cannot handle or neglect or that a CPU exception has been thrown in the kernel while it was in process context. In this case, it is up to the kernel to kill the process concerned with this event. In any case, the main job of the `exit()` syscall is performed by the function `do_exit()`, whose definition appears in <kernel/exit.c>.

This function performs the following activities:

- Enable `PF_EXITING` in the `flags` field of its process descriptor
- The function `del_timer_sync()` is used to tear out any kernel timer
- Update process accounting information, if applicable, i.e. if the kernel was compiled with `CONFIG_BSD_PROCESS_ACCOUNTING` configuration option
- Release or delete the resources related to the process, including process address space, IPC semaphores, file systems, open file descriptors, etc. The action of freeing or destroying depends on whether these resources are shared or not among other processes and it is practically performed by some specific functions which detach or erase the related data structures in the `task_struct` of the process
- Populate the `exit_code` field of the process descriptor with the exit code provided by the application code that called `exit()` or by the kernel itself
- Call `exit_notify()` which in turn notifies the parent process, update the relationships of its childs, if any, by finding new parents for them and set the `exit_state` field of `task_struct` to `EXIT_ZOMBIE`
- Finally call `schedule()` in order to invoke the scheduler

Once the above operations are completed, the task is said to be in a *zombie* state. This means that all the resources associated to it are released and it is not meant to run any more. However, the main memory still accommodate its kernel stack, its `thread_info` and `task_struct` structures, in order to enable the system to retrieve information about the process after its death.

In order to deallocate this memory, the parent that created the now-zombie process must invoke a function from the `wait()`-family syscalls, e.g. `wait()`, `waitpid()` [18, 4]. These functions wrap around the `wait4()` syscall and are meant to suspend the execution of the calling process until one of its children changes state: specifically, this can be a process termination, a stop or resumption by a signal. If the child is terminated, the parent collects information about its child and the function returns its PID on success. Then, the remaining resources are finally released by means of the function `release_task()`.

1.2 Linux process scheduler

1.2.1 Scheduling basics

After analysing the process model and how processes are actually described and handled in Linux, it is now time to dive into the kernel subsystem that decides which tasks, among the runnable ones, have to run, when and how long: the *process scheduler*.

In any operating system, the role of the scheduler is critical since the decision about "who has to run next" can significantly affect the performances of the entire system. Without a scheduler, it would be literally impossible to tune and coordinate the execution of several runnable processes, thus allowing to have a *multitasking* system.

A multitasking system, like any computer or server existing nowadays, by definition allows multiple processes to execute in an interleaving fashion: in a single processor machine, switching the execution of a task in favour of another gives the illusion of the processes running concurrently; in multi-processor machines, this effectively enables the execution of more processes at the same time, i.e. in parallel, on different cores.

Therefore, the scheduler is responsible for assigning resources to the all the runnable processes by defining a proper temporal order for this assignment, in order to maximise the overall performances of the system; this important decision is a direct consequence of the fact that not all the runnable tasks can always run, because, at any given time, there could be more runnable tasks than the number of available cores (which is quite often the case).

Moreover, some processes have the capability to *block* themselves or to *sleep*; this occurs when, at some point during its execution, the running task needs to wait for a certain event to happen in order to advance its execution, e.g. an input from a peripheral, or it was simply designed to perform certain operations periodically.

There are basically two categories of multitasking operating systems: *preemptive multitasking* and *cooperative multitasking*. In the first case, it is up to the scheduler to block the execution of a process and let another task enter the running state, by *preempting* the suspended process in favour of the newly running one; in the second scenario, instead, a process is allowed to run until it voluntary stops running, by *yielding* its execution. Linux belongs to the family of preemptive multitasking systems.

Without going too much into the details, the drawback of the second approach is that, since the operating system cannot perform the global decision of suspending the execution of a task, it can lead to big dreadlocks and, eventually, block the entire system.

As mentioned, in a multitasking system (whatever it is), the scheduler has a great impact on the overall performances. The temporal coordination among runnable, sleeping or waiting processes, and the related state transitions, must be performed in such a way to provide the final user with an effective trade-off between high throughput, i.e. maximise the utilisation factor of the system, and low latency, i.e. minimise the response time of the various processes.

In this regard, it is worth classifying the processes in two fundamental categories: *processor-bound* and *I/O-bound* processes. The former are tasks which spend most of their available CPU time in executing code; their execution is not often constrained by I/O peripherals and they generally run until there is an higher priority task in runnable state, when preemption can take place. An example can be a simulation software or a mathematical solver. Conversely, to the latter category belong those processes which spend most of their time in requesting access to an I/O and waiting for the peripheral response; in this context, the definition of peripheral is extended to any resource that may be contended among processes, e.g. mouse inputs, network resources or disk I/Os. An example can be a Graphical User Interface program.

As a consequence, in the ideal case, processor-bound processes shall be scheduled for longer time but less frequently, while I/O-bound processes demand to be often scheduled for shorter runs. It goes without saying that it is not possible to categorise all the processes as strictly belonging to one of these two fields, since there is a huge grey area between the two and some tasks can show both behaviours.

However, this classification is directly correlated with the performance tradeoff that the scheduler is in charge to find between low latency and high throughput; specifically, a policy action that favours I/O-bound tasks over processor-bound ones will decrease the latency, therefore provide fast response time, at the cost of more frequent context switches, thus decreasing processor utilisation and penalising processor-bound processes.

This is what happens in the Linux scheduler by means of the Completely Fair Scheduler (CFS), which tends to benefit I/O-bound processes but still not neglecting processor-bound processes, providing the user with great interactivity and good system throughput.

In describing a scheduler policy, usually two concepts are fundamental: the timeslice and the priority system.

The scheduler timeslice is defined as the maximum amount of time that a task is allowed to run until it is preempted. Long timeslices leads to decrease interactiveness, favouring processor-bound processes while too short timeslices makes the system waste a lot of processor time in context switches, penalising cache effects and, consequently, processor-bound tasks.

A priority-based scheduler assigns different priority levels to all the tasks in the system, which reflect the importance and the need for processor time of each task.

In Linux OS, two priority ranges are implemented in the kernel. The first range is the *nice* value: this is a numerical value between -20 and +19, with 0 as default. The higher the *nice* value, the lower is the priority of the respective process: the name comes from the funny interpretation that a lower priority process acts more "nicely" with respect to other processes in the system.

The second range of priorities is the *real-time priority*: this range spans over 100 values from 0 to 99. in contrast with nice values, higher real-time priorities reflect greater priorities. It is worth noting observing that these two sets, i.e. nice and real-time priorities, are disjointed, thus all real-time processes have higher priority than standard processes.

While in the first versions of Linux and in other operating systems schedulers, there exists a direct mapping between nice values and timeslices, Linux CFS does not employ this approach. Furthermore, the concept of timeslice is replaced by a *proportion* of the processor. In this context, the amount of processor time allowed to a process is a function of the dynamic load of the system, while the nice value simply behaves as a weighting factor, which is further included in the scheduling algorithm.

1.2.2 Scheduler classes

The Linux scheduler is designed with a modular approach [18, 19]. This means that there are several different algorithms responsible of scheduling different types of processes. Each of these algorithms is implemented in a different *scheduler class* with a different priority.

The code for different modules of the scheduler are located in the <kernel/sched> directory of the Linux source tree. Some important files are the following:

- `core.c` includes the core section of the scheduler and it implements routines used by each scheduling class
- `fair.c` implements the default Linux Completely Fair Scheduler
- `rt.c` contains the module for scheduling real-time tasks
- `idle_task.c` carries out the operation performed by a special Linux task named the *idle task*

The general scheduler code resides in <kernel/sched/core.c>. The entry point of the scheduler is the function `schedule()` and its helper `__schedule()`. These routine are generic with respect to the several scheduler classes and, each time the scheduler is invoked, `__schedule()` calls in turn `pick_next_task()` which iterates over each class in a priority order. The first class with a runnable task is the one invoked, which will schedule the next process to run.

Each scheduling class is implemented as a structure of type `sched_class` whose fields are mostly function pointers pointing to the key routines that needs to be performed by each module as it operates. Some of them are listed in the following snippet of code:

```
struct sched_class {
  const struct sched_class *next;

  void (*enqueue_task) (struct rq *rq, struct task_struct *p,
    int flags);
  void (*dequeue_task) (struct rq *rq, struct task_struct *p,
    int flags);
  void (*yield_task) (struct rq *rq);
  ...
  void (*check_preempt_curr) (struct rq *rq, struct task_struct
    *p, int flags);

  struct task_struct * (*pick_next_task) (struct rq *rq,
                        struct task_struct *prev,
                        struct rq_flags *rf);
  ...
  void (*set_curr_task) (struct rq *rq);
  void (*task_tick) (struct rq *rq, struct task_struct *p, int
    queued);
  void (*task_fork) (struct task_struct *p);
  void (*task_dead) (struct task_struct *p);
```

A brief description of the above fields is the following:

- `enqueue_task()` (or `dequeue_task()`) is called when a task enters (or exits) the runnable state, in order to insert (or remove) the input task in the class's runqueue and increment (or decrement) the runqueue-variable `nr_running`
- `yield_task()` is invoked when a task voluntarily releases the CPU, but still keeping the runnable state

- check_preempt_curr() performs a check in order to determine if the currently running process should be preempted in favour on the newly runnable task
- pick_next_task() is responsible of choosing the most suitable task to run next
- set_curr_task() is invoked when a task changes scheduling class
- task_tick() is periodically called from the scheduler tick at each timer interrupt in order to update scheduling variables and possibly switch a running task
- task_fork() (task_dead()) is used to notify when a task is created (dies)

Based on the scheduling policy assigned to a task, the kernel establishes which scheduling class shall be assigned to. Processes belonging to SCHED_NORMAL, SCHED_BATCH and SCHED_IDLE are mapped to the CFS class fair_sched_class while tasks labelled with SCHED_RR and SCHED_FIFO policy falls into the real-time scheduler class rt_sched_class

Apart from the CFS and real-time schedulers, there exists also other special classes which implements different algorithms, which can be enabled by the user on demand in order to schedule some special task according to the specific need. In this thesis, however, we will focus on the CFS class, implemented in fair_sched_class, which is the default scheduler class for standard processes introduced in kernel version 2.6.23.

```
const struct sched_class fair_sched_class = {
  .next            = &idle_sched_class,
  .enqueue_task    = enqueue_task_fair,
  .dequeue_task    = dequeue_task_fair,
  ...
  .pick_next_task      = pick_next_task_fair,
  ...
  .task_tick       = task_tick_fair,
  ...
  .update_curr     = update_curr_fair,
  ...
};
```

1.2.3 Fair scheduler model

The CFS algorithm relies on the approximation of an ideal multitasking model [18, 19, 20], i.e. a system with a perfectly multitasking processor. In such an ideal system, all the runnable processes would run concurrently using $1/n$ the total processor's power. Unfortunately, this model is impracticable, since such a processor simply does not exist.

However, following this concept, a first approximation to this ideal behaviour, could be that all the runnable processes would run for the same amount of time, using all the processor's power: this time would be the total processor's time divided by the number of runnable tasks n in the system, thus *fairly* sharing the total runtime among all the tasks. Furthermore, in this still ideal scenario, the scheduler would schedule all the runnable tasks for infinitely small periods, so that, considering any time interval, all the processes would have received the same amount of CPU time.

It goes without saying that this is not feasible too, since running tasks for too small timeslices would significantly degrade performances. This is due to the high cost of switching context between one process and another so often. The context switch per-se is a critical operation from a performance standpoint, since the processor has to store the state of the exiting task, i.e. by updating the process descriptor, and either loading another new process in memory or resuming a previously stopped execution; this strongly impacts the probability for the processor to find useful data or instructions in the cache memory, therefore reducing the performances.

CFS does not assign proper timeslices to the processes in order to determine how much they should be allowed to run: instead, CFS computes how long a task should run as a function of the dynamic load of the system [18, 19, 20], i.e. the number of total runnable tasks each time it is invoked. Therefore, the previous notion of timeslice is substituted with the fraction of processor's time assigned to a process, in order to express the execution time allotted to each task before being possibly preempted.

The role played by the *nice* value is different as well: in CFS there is no more direct mapping between timeslices and nice values [18], as it used to be in previous kernel versions; conversely, now the nice value is instead used to weight the fraction of processor time that each process shall receive. Higher priority tasks (with lower nice value) receive larger weights, leading to more CPU-time.

Each task is run for a period that is proportional to its weight divided by the total weight of all the runnable tasks. In order to deal with a real not perfectly multitasking system, CFS sets two important parameters resulting in a pair of scheduling constraints:

- *Targeted latency*: lower bound on the minimum scheduling period in which all runnable tasks are scheduled, at least once
- *Minimum granularity*: lower bound on the minimum time a task is allowed to run, before being preempted

The first value is used to approximate an infinitely small scheduling period; in particular, smaller values for the targeted latency leads to better interactivity at the cost of necessarily increasing the number of context switches, yielding to less system throughput; its default value is usually set to 20 ms. The second parameter comes from observing that, as the number of runnable tasks increase, the timeslice assigned to each process would tend to zero. Therefore, the minimum granularity is usually set to 1 ms in order to leverage this scenario by preventing the OS from assigning too small timeslices, thus introducing other unsustainable switching penalties.

It is worth observing that the *fairness* of the algorithm decreases as the number of runnable tasks grows too much, that is when the proportion of CPU time assigned to each process is rounded down to the value of the minimum granularity. Furthermore, when the number of tasks grows significantly up to the point that the default targeted latency is not enough to schedule all the tasks, it is progressively increased in step of 4 ms. Nevertheless, for a reasonable number of running tasks, CFS performs a great job in terms of scheduling fairness.

1.2.4 Completely Fair Scheduler implementation

After describing the logic and the basic concepts behind CFS, it is now possible to dive into the scheduler implementation. This is fundamental to understand how the scheduler actually works and, eventually in this thesis, describe the introduced modifications.

Scheduling entity

Apart from single tasks, the default Linux scheduler CFS is capable of scheduling also group of tasks. More in general, CFS deals with the extended concept of scheduling entities. A scheduling entity is described by means of the data structure `sched_entity`, defined in <include/linux/sched.h>.

This structure incapsulates all the scheduling-relevant information regarding load, time accounting, runqueue and statistics:

```
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;
  u64       prev_sum_exec_runtime;

  u64       nr_migrations;

  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
...
};
```

Some of the most important fields are here described:

- `load`: weight factor based on the system load
- `run_node`: a node of the red-black tree describing the runqueue
- `on_rq`: flag that specifies whether the node belongs to a runqueue or not
- `exec_start`: variable storing the timestamp of the execution start for a task
- `sum_exec_runtime`: variable storing the total execution time for a task
- `vruntime`: key CFS scheduling parameter; it is the actual runtime weighted by the load of the queue of runnable processes
- `prev_sum_exec_runtime`: variable storing the previous runtime of a task

This structure is placed inside the process descriptor of each task as the field `se` of the structure `task_struct` and it is therefore accessible from the current running process.

Before going further with the description of CFS implementation, it is worth spending some words on the member `load` of structure `sched_entity`, since it is

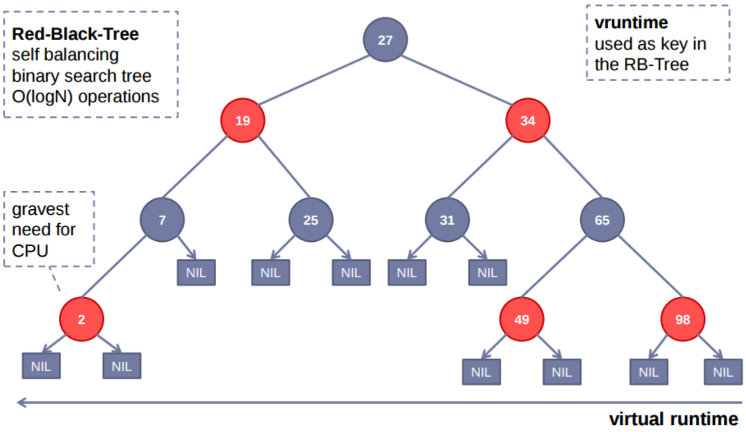


FIGURE 1.5: CFS red-black tree.

directly involved in the update of the key parameter `vruntime`. This field is a load weighting factor depending on the type and the static priority of a task. The structure `struct load_weight`, defined in <include/linux/sched.h>, includes both its value and its reciprocal:

```
struct load_weight {
  unsigned long      weight;
  u32        inv_weight;
};
```

This load factor is correlated with the static priority, i.e. `nice` level, of the task; therefore it directly enters in the update of `vruntime`. In order to compute this number, the kernel uses a table defined in <kernel/sched.c> in order to map all 40 nice levels, i.e. from -20 to +19, to a suitable weighting factor:

```
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,
};
```

Without going too much into details, these values are meant to give ±10% of CPU time to a task whose priority is increased or decreased of one nice level down or up, respectively [19].

Red-black tree

One of the most interesting characteristics of the CFS design concerns the implementation of the runqueues; instead of using the traditional data structures meant to describe the runqueues, CFS employs a *red-black tree*, aimed at generating a timeline of processes scheduled for running [20]. Its code is placed in <include/linux/rbtree.h>.

A red-black tree (RBTree), as the one depicted in figure 1.5 belongs to the family of self-balancing binary search trees. The name comes from the fact that the nodes have also a colour attribute, that is they can be either red or black. Furthermore, naming as *branch* or *leaf* nodes the tree nodes with or without children, respectively, the following six constraints must be satisfied for an RBTree:

1. Each node is either red or black;

2. All leaves (nodes with no children) are black;

3. All leaves do not contain data;

4. All branch nodes (nodes with children) have two children;

5. If a node is red, its children are black;

6. For a given node, each path to any of its leaves encounters the same number of black nodes.

These constraints guarantee that the depth (distance from the root node) of the farthest leaf is at most double the depth of the highest leaf in the tree [18].

Since CFS requires a lot of time accounting in order to properly perform its job, this kind of data structure is particularly well-suited because it allows to efficiently perform insert and remove operations, that are executed in O(log N) time, where N is the total number of nodes in the tree.

The leafs of the RBTree are organised in a hierarchic fashion, based on the key scheduling parameter of CFS: the *virtual runtime*, whose value is stored in the `vruntime` variable. As processes advance their execution, this variable is updated and a new node is inserted in the tree. The smaller is the `vruntime`, the more the corresponding scheduling entity is placed to the left of the tree, as shown in figure 1.5.

The RBTree data structure is implemented in the `struct cfs_rq`, which is in turn placed into each per-CPU base runqueue `struct_rq`. Without discussing the general runqueue data structure, we focus our attention on `struct cfs_rq`, defined in <kernel/sched/sched.h>:

```
struct cfs_rq {
  struct load_weight load;
  unsigned int nr_running, ...;
  ...
  u64 min_vruntime;
  ...
  struct rb_root_cached tasks_timeline;
  /*
  * 'curr' points to currently running entity on this cfs_rq.
  * It is set to NULL otherwise (i.e when none are currently running).
  */
  struct sched_entity *curr, ...;
  ...
};
```

The most important fields with respect to the analysis carried out in this thesis are:

- `load`: cumulative load weighting factor of tasks in the runqueue

- `nr_running`: number of runnable tasks of the runqueue

- `min_vruntime`: monotonically increasing variable storing the minimum `vruntime` of the runqueue. This variable is used as benchmark to compare all the tasks against and select the next task eligible to run

- tasks_time_line: this structure gives access to both the root and the left-most node of the runqueue
- curr: pointer to the scheduling entity of the currently running task

Before going further with the description of the core CFS implementation, it is worth pointing out that all the functions referred in the next sections, if not specified, will be assumed to reside in <kernel/sched/fair.c>.

Virtual Runtime

As mentioned, the vruntime has a crucial role in the CFS algorithm. It tries to approximate the underlying idea of perfectly multitasking processor; in such an ideal processor, its value would be constant for all same-priority tasks indeed, since all processes would run concurrently and, therefore, they would always receive the same amount of CPU time. In a real scenario, instead, the vruntime is introduced in order to express how long a process has already run and how much longer it is entitled to run.

Almost every time the scheduler is involved, the function update_curr(), is called in order to update this time accounting for the currently running task:

```
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);
    cpuacct_charge(curtask, delta_exec);
    account_group_exec_runtime(curtask, delta_exec);
  }

  account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

First the execution time of the current process is computed and the value is stored in delta_exec. Then, this value is passed to calc_delta_fair, which actually performs the bulk of the work: it normalises this runtime with respect to the other runnable processes and computes the variation of vruntime to be eventually added to the vruntime of the current process.

Apart from some rounding and bit-shifting magic, this variation is essentially computed as:

```
delta_exec_weighted = delta_exec*(NICE_0_LOAD/curr->load.weight)
```

As a result, more important tasks, i.e. those with lower nice value, will receive a larger weight, thus a smaller vruntime will be assigned to them, moving them to the left side of the RBTree and increasing their probability to be picked up for running. As a matter of fact, for tasks at nice level 0, curr->load-weight is equal to NICE_0_LOAD, resulting in a unitary weighting factor, for which the computed variation of vruntime coincides with the physical elapsed execution time of the task.

Finally, the next operations are to update the per-queue min_vruntime as well as some additional accounting. The variable min_vruntime is updated by calling the function update_min_vruntime(). When this is done, the kernel first checks if the tree has a leftmost element: if yes, its vruntime, which is also the smallest in the tree, is obtained; otherwise, if the tree is empty, the vruntime of the current process is selected. In order to ensure that min_vruntime always increase monotonically, its value is finally set to the maximum between the vruntime just chosen and what was already set in the runqueue. In this way, the actual value in the runqueue is updated only if the vruntime of one of the nodes in the tree is greater than its previous value, thus guaranteeing that min_vruntime always increases.

Pick the next task

Once all the necessary assumptions have been made and the involved data structures have been described, the CFS scheduling algorithm can be summarised as simply "select the leftmost node of the tree, which represents the task with the smallest vruntime".

This selection is performed in general by the function pick_next_entity(), but specifically the actual pick of a scheduling entity from the tree is done by the helper routine __pick_next_task():

```
static struct sched_entity *__pick_next_entity(struct
    sched_entity *se)
{
  struct rb_node *next = rb_next(&se->run_node);

  if (!next)
    return NULL;

  return rb_entry(next, struct sched_entity, run_node);
}
```

This function calls in turn rb_next() which implements the actual logic of efficiently going through the tree and return the leftmost node.

Now that the task is selected, some additional work needs to be done in order to make it the actual running task. The function set_next_entity() is meant to perform this job:

```
static void
set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
```

```
{
  /* 'current' is not kept within the tree. */
  if (se->on_rq) {
    ...
    update_stats_wait_end(cfs_rq, se);
    __dequeue_entity(cfs_rq, se);
    update_load_avg(se, UPDATE_TG);
  }

  update_stats_curr_start(cfs_rq, se);
  cfs_rq->curr = se;
  ...
  se->prev_sum_exec_runtime = se->sum_exec_runtime;
}
```

Apart from updating statistics and accounting as usual, this function removes the task from the tree, since it is about to become the currently executing one, by invoking `__dequeue_entity()`. Then, even if the process has been just removed from the tree, the CFS runqueue needs to be updated in order to reflect its new currently running process. Finally, the field storing the previous runtime of the process is updated with the cumulative runtime value stored in `sum_exec_runtime`; since `sum_exec_runtime` is not reinitialised, the difference between `sum_exec_runtime` and `prev_sum_exec_runtime` is the actual time the task has been running on a CPU.

Scheduler tick and preemption

Apart from specific cases when a system call is executed, e.g. a process that forks a child or a process that terminates, the scheduler is also periodically invoked. This event, called scheduler tick, is handled by a timer and is responsible of checking if the running task needs to be preempted in favour of another task.

At each scheduler tick, the function `task_tick_fair()` is executed and delegates most of the work to the function `entity_tick()`. This first calls `update_curr()` in order to update all the scheduling parameters, then it checks the `nr_running` field of the CFS runqueue: if there is only one runnable process in the queue, no operation needs to be performed, since there is no potential task that could preempt the currently running one. Otherwise, the preemptive action is handled by the function `check_preempt_tick()`:

```
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));
    ...
    return;
  }
  ...
}
```

The core of this function is to prevent a task from running longer than the fraction of latency period assigned to it, i.e. its timeslice. To this purpose, the function computes this timeslice by calling sched_slice() and then it also calculates how long it has been running as the difference between sum_exec_runtime and prev_sum_exec_runtime. If the task has been running longer than the timeslice, resched_task() comes into play in order to mark that the main function __schedule() needs to be called in order to preempt the current running task in favour of another runnable one.

As mentioned, the actual dynamic timeslice is computed in the function sched_slice():

```
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
  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);
  }
  return slice;
}
```

This function first computes the scheduling period, within all the runnable tasks are promised to run at least once, in the function __sched_period():

```
static u64 __sched_period(unsigned long nr_running)
{
  if (unlikely(nr_running > sched_nr_latency))
    return nr_running * sysctl_sched_min_granularity;
  else
    return sysctl_sched_latency;
}
```

In the previous sched_slice(), the above function is called passing as argument the number of tasks already in the runqueue plus the currently running one, in case it was previously removed from it. By default, the scheduling period is set to sysctl_sched_latency, meant for scheduling a standard number of sched_nr_latency processes for at least the minimum granularity, whose value is stored in sysctl_min_granularity. If the number of runnable tasks exceeds the default sched_nr_latency, then the scheduling period is set to the minimum granularity multiplied by the number of ready tasks. All these time values are intended in time units of *ms*.

Coming back to the sched_slice() routine, after updating all scheduling entities and runqueue loads, the function __calc_delta() is invoked to finally get the actual timeslice for the currently running task. This value, apart from some bit-shifting magic, is essentially calculated as:

$$\Delta t_{slice} = \Delta t_{latency} \times (W_{curr}/W_{tot}) \tag{1.1}$$

where $\Delta t_{latency}$ is the previously computed scheduling period while the two weighting factors W_{curr} and W_{tot} are `se->load-weight` and `load` variables in the function `sched_slice()`, respectively.

Insert a new task into the tree

When a process is created by means of a call to `fork()`, or also when it is woken up and re-enters the runnable state, CFS needs to put it into the RBTree. This job is done in the function `enqueue_entity()`.

```c
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se,
    int flags)
{
  bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags &
    ENQUEUE_MIGRATED);
  bool curr = cfs_rq->curr == se;
  ...
  if (renorm && curr)
    se->vruntime += cfs_rq->min_vruntime;

  update_curr(cfs_rq);
  ...
  if (renorm && !curr)
    se->vruntime += cfs_rq->min_vruntime;
  ...
  update_load_avg(se, UPDATE_TG);
  enqueue_entity_load_avg(cfs_rq, se);
  update_cfs_shares(se);
  account_entity_enqueue(cfs_rq, se);

  if (flags & ENQUEUE_WAKEUP)
    place_entity(cfs_rq, se, 0);

  check_schedstat_required();
  update_stats_enqueue(cfs_rq, se, flags);
  check_spread(cfs_rq, se);
  if (!curr)
    __enqueue_entity(cfs_rq, se);
  se->on_rq = 1;

  if (cfs_rq->nr_running == 1) {
    list_add_leaf_cfs_rq(cfs_rq);
    check_enqueue_throttle(cfs_rq);
  }
}
```

Basically this function first renormalises the `vruntime` of the inserted task by adding the `min_vruntime` of the CFS runqueue to it (otherwise it would not be consistent to compare the values, since `minvruntime` always increases monotonically) and calls `update_curr()`. Then, after updating loads and adding them to the CFS runqueue, the function distinguishes two cases:

- If the process has been sleeping, the function `place_entity()` needs to be called in order to adjust its `vruntime` before actually inserting the task in the tree by means of the helper `__enqueue_entity()`

- If the task has been running before, the helper `__enqueue_entity()` can be called directly in order to insert it into the RBTree; this insertion is safe and the process is put in the right position, as assured by the previous updates of `vruntime` and `min_vruntime`

Without going too much into details, what `place_entity()` does can be summed up as follows: since the kernel must schedule all the runnable processes such that they will run at least once in the current latency period, this function makes sure that the newly awoken task will run only after the current latency period expires.

Remove a task from the tree

When a process stops its execution, e.g. it blocks or goes to sleep, or also when it terminates, CFS needs to remove its scheduling entity from the RBTree. This operation is performed by the function `dequeue_entity()`:

```
static void
dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se,
   int flags)
{
  /*
  * Update run-time statistics of the ’current’.
  */
  update_curr(cfs_rq);
  ...
  update_load_avg(se, UPDATE_TG);
  dequeue_entity_load_avg(cfs_rq, se);

  update_stats_dequeue(cfs_rq, se, flags);

  clear_buddies(cfs_rq, se);

  if (se != cfs_rq->curr)
    __dequeue_entity(cfs_rq, se);
  se->on_rq = 0;
  account_entity_dequeue(cfs_rq, se);
  ...
  if (!(flags & DEQUEUE_SLEEP))
    se->vruntime -= cfs_rq->min_vruntime;

  /* return excess runtime on last dequeue */
  return_cfs_rq_runtime(cfs_rq);

  update_cfs_shares(se);
  ...
}
```

The function performs the following main operations:

1. Update scheduling parameters by calling `update_curr()` as usual

2. Update both the loads for the scheduling entity of the dequeued task and for the CFS runqueue, to have them synchronised to the current time

3. Subtract the load of the scheduling entity from the load of the CFS runqueue

4. Subtract also its weight from the the weight of the runqueue

5. Call the helper `dequeue_entity()`, which actually removes the scheduling entity from the RBTree

6. If the task is being removed not because it went to sleep, normalise its `vruntime` by subtracting the `min_vruntime` of the runqueue from it

7. Update the runqueue `min_vruntime` only if the task is being removed not because it went to sleep

Chapter 2

Performance counters on multi-core processors

2.1 Multi-core processors

Modern processors nowadays are asked to provide computer systems with always more performances and computing capabilities, in order to satisfy the demand coming from the most advanced scientific and technical challenges.

In the first generations of Intel Pentium processors as well as the first ARM and AMD processors, this was achieved by simply increasing the processor's clock frequency, intuitively resulting in faster processing units, capable of executing more instructions per unit time.

On the other hand, thanks to the rapidly increasing transistors fabrication, following Moore's Law, and their constantly decrease in size, the trend to achieve higher clock speeds became to put together billions of transistors on chip which, in theory, should result in more powerful and faster processing units. This of course has brought down some drawbacks, that turn out to be not negligible at all.

First of all, cramming so many transistors on the same chip and continuously switching them on and off at rates in the order of thousands of times per second, implies essentially overheating problems. As a consequence, suitable cooling systems had to be developed, aimed at handling this dissipated power, in order to prevent high precision and high clock frequencies from being killed, or literally *burnt out*, by the generated heat.

Furthermore, running all these transistors together, and thus raising the clock frequency up, also produces a significant voltage growth which in turn makes the chip drastically consume more power.

For these reasons, another path has been followed, in order to reach the higher performances of late computer systems, which was to put more processing units, i.e. cores, in parallel, thus giving birth to the multi-core processors we have nowadays.

2.2 Performance monitoring hardware overview

Due to the intrinsic correlation between computers evolution and their performance orientation, most important processors producers have started building and adding special hardware into their multi-core processors, whose primary purpose is to monitor the performance of the system as well as of user-space applications, by means of simple and effective count measurements. These special hardware units, that most CPUs nowadays are equipped with, are named as Performance Monitoring Counters (PMCs).

These units are structured as a set of special-purpose hardware registers embedded into each core of the multi-core processor. In general, they allow to monitor a particular hardware activity related to the respective processor they belong to; moreover, this is performed by measuring, literally *counting*, the occurrences of one or more pre-programmed events. Some examples of events include, among others, the number of cache-misses for a specific cache level, the number of instructions or how many CPU cycles have been counted during the execution of a specific portion of code.

In order to enable these measurements, a minimal PMC unit must include at least two types of HW registers per core:

- *event selection registers*, which are meant to be programmed by writing on their not reserved bits. They allow to specify which events are to be monitored and to properly configure them

- *performance counter registers*, which store the counting of the programmed events and are meant to be read in order to retrieve the measured values

Apart from these two fundamental kinds of registers, modern PMC units are also equipped with some *global registers*, which allow to specify additional global counters configurations, register overflow occurrences, debug, and other features.

It goes without saying that the size of the set of measurable events is bounded by the availability of these counters on the underlying hardware, first of all their number, which tends to vary among the different microprocessors architectures. Without loss of generality, PMCs availability and the functionalities they may provide are essentially an architecture dependent feature of modern processors.

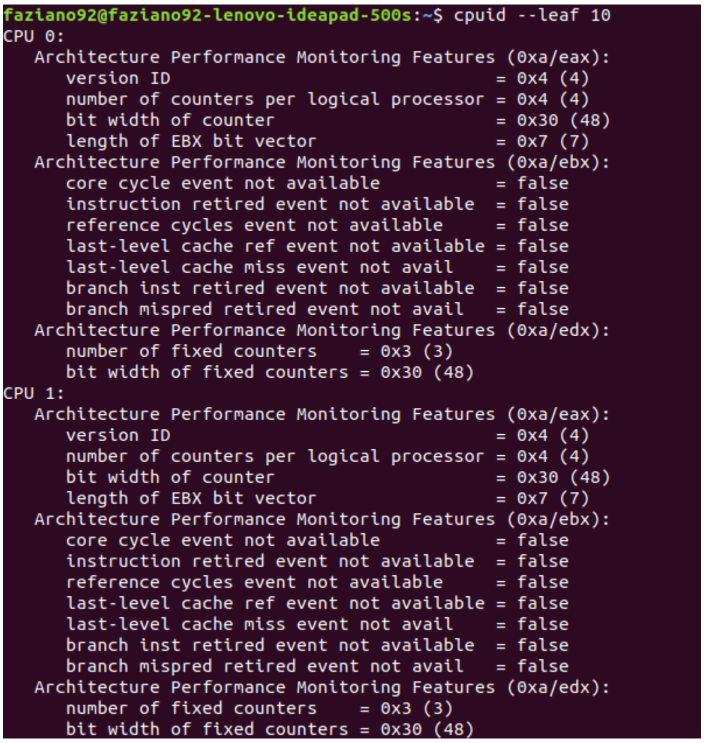
2.3 Intel architecture

Intel introduced performance monitoring units on its hardware starting from the Pentium architecture, as a set of model-specific registers (MSRs) [8]. These registers were primarily added in order to help application and system programmers address applications and system performances by monitoring processor performance parameters of interest.

The technology was further enhanced with the Intel P6 family and the Intel NetBurst microarchitectures, in order to allow broader set of events to be selected and provide with better control features over them. Starting from more recent architectures, such as Intel Core Solo and Intel Core Duo, Intel started distinguishing and considering two types of performance events: *architectural* and *non-architectural* events. Both types of events are based on a counting or interrupt-based sampling mechanism; the difference is that events belonging to the former class persist across different architectures while those provided by the latter may, and often do, change across different processor models.

Therefore, architectural performance events class guarantees backward compatibility among different architectures, thus allowing the programmer to easily reuse the same counters configuration and settings to gather information and implement cross-tests between different micro-processors, with almost zero effort in switching from a model to another. Of course this comes with a cost, which is in general a smaller set of monitorable architectural events with respect to their non-architectural hardware-specific counterpart.

Another important difference between the two is that the availability of architectural performance events on a given platform is definitely easier to check with



```
faziano92@faziano92-lenovo-ideapad-500s:~$ cpuid --leaf 10
CPU 0:
   Architecture Performance Monitoring Features (0xa/eax):
      version ID                               = 0x4 (4)
      number of counters per logical processor = 0x4 (4)
      bit width of counter                     = 0x30 (48)
      length of EBX bit vector                 = 0x7 (7)
   Architecture Performance Monitoring Features (0xa/ebx):
      core cycle event not available           = false
      instruction retired event not available  = false
      reference cycles event not available     = false
      last-level cache ref event not available = false
      last-level cache miss event not avail    = false
      branch inst retired event not available  = false
      branch mispred retired event not avail   = false
   Architecture Performance Monitoring Features (0xa/edx):
      number of fixed counters    = 0x3 (3)
      bit width of fixed counters = 0x30 (48)
CPU 1:
   Architecture Performance Monitoring Features (0xa/eax):
      version ID                               = 0x4 (4)
      number of counters per logical processor = 0x4 (4)
      bit width of counter                     = 0x30 (48)
      length of EBX bit vector                 = 0x7 (7)
   Architecture Performance Monitoring Features (0xa/ebx):
      core cycle event not available           = false
      instruction retired event not available  = false
      reference cycles event not available     = false
      last-level cache ref event not available = false
      last-level cache miss event not avail    = false
      branch inst retired event not available  = false
      branch mispred retired event not avail   = false
   Architecture Performance Monitoring Features (0xa/edx):
      number of fixed counters    = 0x3 (3)
      bit width of fixed counters = 0x30 (48)
```

FIGURE 2.1: Example of `cpuid` output.

respect to non-architectural ones. In particular, the number of available counters and the monitorable events can be easily queried by means of the respective `CPUID` instruction for the processor in question. In x86 architectures, if supported, this instruction returns processor identification as well as feature information and HW details in the `EAX`, `EBX`, `ECX` and `EDX` registers. As far as the PMU is concerned, the output of `CPUID` contains also detailed data, such as number of counters and their bit width, and also which events are available on the platform. Despite `CPUID` being a quite low-level instruction, it is still easy to use it and it can be called from higher level contexts. As a matter of fact, it is possible to return the portion of its output concerning the available PMU units, in a nice and human readable format, by simply executing the Linux command `cpuid` with the `leaf` option set to 10.

Figure 2.1 shows the output of the above command, relative to the first two (out of four) cores of a machine with an Intel Core i5-6200U quad-core processor. The lines of output for the last two cores are omitted for brevity they are the same as the first two, since the considered machine is an SMP one (Symmetric Multi-Processing).

As mentioned, architectural performance monitoring facilities persist across processor implementations; moreover, there exist four versions (specified in the ID field of the `cpuid` output) of architectural PMC units: higher version IDs correspond to newer and advanced technology adopted together with more available HW to further configure and control the registers involved in the counting.

As far as Intel is concerned, configuring an architectural PMU to count a specific event boils down to properly programming the event select MSRs, named as `IA32_PERFEVTSELx`, where "x" denotes the ID (0, 1, 2, ...) of the relative core. Each of these registers is paired with a performance monitoring counter MSR, labelled as `IA32_PMCx`.

Figure 2.2 shows the bit field layout of the `IA32_PERFEVTSELx` register for machines supporting architecural performance monitoring technology version 3 and 4.

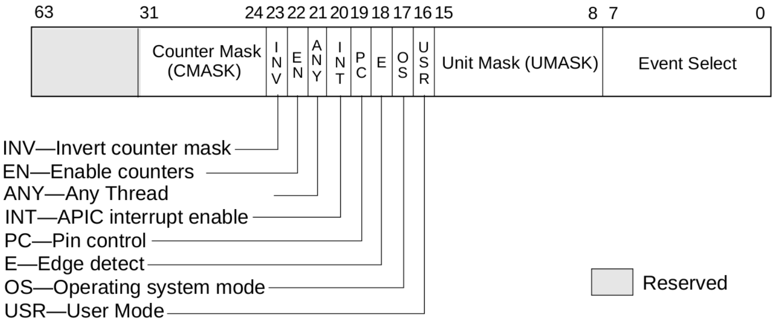


FIGURE 2.2: Bit field layout for IA32_PERFEVTSELx MSRs.

The attribute "architectural" refers specifically to the fact that both bit field layout and addresses of IA32_PERFEVTSELx MSRs, as well as the IA32_PMCx addresses, are the same across microprocessors. In particular, it is worth describing the bit fields of the IA32_PERFEVTSELx register:

- **Event select (bits 0-7):** select the event logic unit employed to reveal a certain micro-architectural condition; in order to monitor a specific event, this value is paired with a suitable UMASK field value. Some of the pre-defined values might not be supported for a given processor

- **Unit mask UMASK (bits 8-15):** select the condition in which the selected event shall be detected; for each architectural event, this value defined a specific micro-architectural condition. This value is paired with a suitable value of the event select bits when monitoring a specific event

- **User-mode flag USR (bit 16):** specify that the selected event is counted when the core is operating in user mode

- **OS-mode flag OS (bit 17):** specify that the selected event is counted when the core is operating in kernel mode

- **Edge detect flag E (bit 18):** enable edge detection for the desired micro-architectural condition

- **Pin control flag PC (bit 19):** when this bit is set, the processor toggles some special internal pins (PMi pins) and increments the counter when a performance events occurs. conversely, if the bit is clear, the processor toggles the same pins on counter overflow. The purpose of these special pins varies from one specific processor to another

- **APIC interrupt enable flag INT (bit 20):** if this bit is set, the processor issues its Advanced Programmable Interrupt Controller to send an interrupt on counter overflow

- **Any Thread flag ANY (bit 21):** when set to 1, this bit enables counting the specified event when it occurs on any logical processor of a core. It is useful to detect the occurrences of an event in when the micro-processor is operating in Intel Hyper-Threading mode, i.e. enabling two logical processors sharing a same physical core

- **Enable counters flag EN (bit 22):** enable/disable performance counting for the relative performance-monitoring counter

Bit position	Event name	UMASK	Event select
0	UnHalted Core Cycles	0x00	0x3C
1	Instruction Retired	0x00	0xC0
2	UnHalted Reference Cycles	0x01	0x3C
3	LLC Reference	0x4F	0x2E
4	LLC Misses	0x41	0x2E
5	Branch Instruction Retired	0x00	0xC4
6	Branch Misses Retired	0x00	0xC5

TABLE 2.1: List of Intel pre-defined architectural events.

- **Invert flag INV (bit 23):** invert the counter-mask CMASK, in order to perform also "less than" comparisons between the counter values and the CMASK; when CMASK threshold is set to zero, this bit is neglected

- **Counter mask CMASK (bits 24-31):** this mask is used to set a threshold value to be compared against the performance counter value; if the event count is greater than or equal to the CMASK value set, than the counter is incremented by one.

As described, a proper combination of values for "Event select" and "Unit mask" fields needs to be specified in order to setup a particular event to be monitored by the PMC unit. Furthermore, table 2.1 summarises all the available pre-defined architectural events as well as the corresponding pair of values of the first two bit fields in `IA32_PERFEVTSELx` that is used to set them in the configuration register.

A short description of all Intel architectural HW events is here reported:

- **Unhalted Core Cycles:** number of core clock cycles counted when the considered core is not in *halted* state (`HALT`). A core is said to be in halted state when its clock signal is not running, thus leaving the CPU in an idle state until an interrupt is received. This event includes state transitions, therefore core cycles are counted also at different core clock frequencies

- **Instructions Retired:** number of instructions that have left the "retirement unit". The "Retirement Unit" is where the results of speculatively executed instructions are written into user-visible registers and the corresponding instructions are removed from the reorder buffer [3].

- **UnHalted Reference Cycles:** number of reference clock cycles counted at fixed frequency while core clock signal running. It does not contemplate performance state transitions, leading to core frequency changes

- **Last Level Cache References:** number of requests from the core that references a cache line in the Last Level Cache (LLC). It is speculative since it might include cache line fills of the first-level cache prefetcher

- **Last Level Cache Misses:** number of cache miss conditions for references to the LLC. It is speculative for the same reasons of the previous Last Level References event

- **Branch Instructions Retired:** number of branch instructions having left the Retirement Unit. It is counted with reference to the retirement of the last micro-operation of the branch instruction

- **All Branch Mispredict Retired:** number of mispredicted branch instructions at retirement. It is counted with reference to the retirement of the last micro-operation of the branch instruction in the architectural path that was mispredicted by the branch prediction HW

Not all these HW events may be supported by a processor equipped with architectural performance monitoring technology and the availability of each of them must always be checked in advance, by executing the `cpuid` command. However, most of the time, these pre-defined events are supported, thus it is possible to count their occurrences on most Intel architectures, while this is definitely not true for non-architectural events, whose availability strongly depends on the specific microprocessor considered.

In the transition from architectural monitoring version ID 1 to version ID 4, several enhancements have been introduced, as additional configuration registers and other features, whose the most important ones are here roughly listed:

- Fixed-function performance counter registers and related control register
- Global control and status registers used for enable/disable counting, overflow counting and overflow clearing
- Enhancement of `IA32_PERFEVTSELx` with the introduction of the AnyThread bit field

Intel non-architectural performance monitoring facilities, as mentioned, provide a wide choice of monitorable events but their usage is constrained by their strong HW specificity, reducing their portability to only machines with similar-architectures. For these reasons, they can be used to focus the eye on a single type of processor, while conducting cross-architecture tests does require additional effort on the programmer side, who has to separately and differently program a PMC unit for each considered architecture. This is where higher-level SW interfaces to access performance counters come along.

2.4 SW interfaces to access performance counters data

Several SW tool and APIs are available to retrieve performance counters data from high level context. In particular, the goal of these libraries is to simplify the access to this dedicated HW, by abstracting the low-level functionalities and details of programming and configuring the PMC units to monitor a specific event. Thanks to these interfaces, it is possible to monitor per-thread or per-core values of performance counters associated to a given set of events. They also generally allow to easily check the available performance counters and events on the running machine by means of command line tools and/or APIs. In this thesis we will focus on the Performance Application Portable Interface (PAPI), a library providing performance counters configuration and access facilities from user-space applications, and PMCTrack tool, which also allows to gather performance counters values from kernel-space.

2.4.1 PAPI

PAPI is an open source cross-platform application programming interface written in C language, whose purpose is to provide a standard API for accessing performance

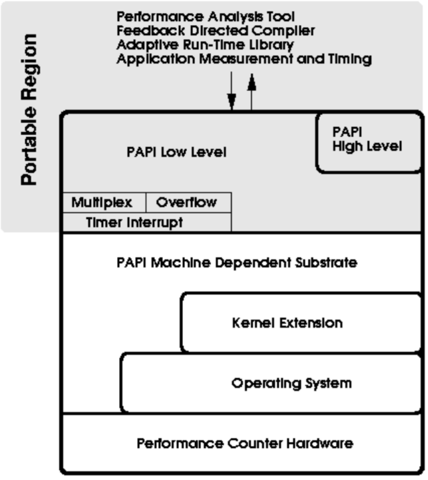

Performance Analysis Tool
Feedback Directed Compiler
Adaptive Run-Time Library
Application Measurement and Timing
Portable Region
PAPI Low Level
PAPI
High Level
Multiplex
Overflow
Timer Interrupt
PAPI Machine Dependent Substrate
Kernel Extension
Operating System
Performance Counter Hardware

FIGURE 2.3: PAPI architecture.

counters HW available on most modern micro-processors [24, 5]. Essentially PAPI generalises the access to performance counters data with respect to the architecture, by providing two main interfaces to the underlying PMC units:

- **High-level interface:** simpler but more limited API, used to perform simple measurements
- **Low-level interface:** more complex but fully programmable API

Hardware events are managed in user defined groups, called *EventSets*[cit need]. The high-level interface allows to simply start, stop and read the counters associated with the list of events contained in the specified EventSet. PAPI includes also a set of predefined events; each preset event is mapped into one or more native events on each hardware platform. Most common performance counter events are found as predefined events.

PAPI architecture is composed by different layers, as shown in figure 2.3. The high and low level interfaces as well as utilities to perform overflow and timer interrupts handling and to multiplex events, are at the top of the architecture, in the machine-independent layer. This portable region calls in turn the *PAPI Machine Dependent Substrate* which handles the machine-dependent code sections of the above layer functions. More on PAPI design and architecture details can be found in [cit]

In order to allow access to the performance monitoring counters at the user-level, the OS needs to be equipped with a device driver which implements functionalities to initialise, start, stop and read the counter. PAPI makes use of the underlying Linux tool *perf*, which adds support to the Linux kernel for accessing performance monitoring counters found in x86 architectures. The `perf` tool abstracts per-process counters data, by introducing a set of *virtual counters* associated to each Linux process, which perceives the counters as private to it and unrelated to other processes different from itself.

The functions of most interest from high-level PAPI are the following ones:

- `PAPI_num_counters(void)` : initialise the library
- `PAPI_start_counters(int *events, int array_len)` : start counting the specified events
- `PAPI_read_counters(long long *values, int array_len)` : copy current counters counts to *values* and reset counters

- `PAPI_stop_counters(long long *values, int array_len)` : stop counting and copy current counters counts to *values*
- `PAPI_accum_counters(long long *values, int array_len)` : add current counters counts to *values* and reset counters

Despite being an effective and easy-to-use tool to configure performance counters and access their data, PAPI is only available in user space. This implies that it is possible to use it to monitor user space processes, i.e. applications, while it does not allow to obtain the same insights at kernel level nor monitoring kernel threads. Due to this intrinsic constraint, it is not possible to use PAPI to access PMCs from kernel space and use their values from the scheduler; its usage is therefore limited to only monitoring user space processes.

2.4.2 PMCTrack

PMCTrack is an open source tool for the Linux kernel which provides an architecture-independent mechanism, allowing the OS scheduler to access per-thread PMC data [21, 14]. It is essentially composed by the following elements:

- PMCTrack Linux kernel patch
- Monitoring modules
- *libpmctrack*
- Command-line tools
- GUI application

PMCTrack access to PMC data relies on the *monitoring module* abstraction: this is the platform dependent component, which is in charge of retrieving the desired high-level metrics to feed the OS scheduler with. In this way, PMCTrack enables programmers to implement architecture-independent scheduling algorithms relying on PMC data [cit need]. However, on their side, programmers in turn have to implement their own architecture-dependent monitoring module, where the events to be monitored are specified as well as the counters are setup.

Figure 2.4 shows the interaction between the OS scheduler and the monitoring module. The main point is that the scheduler is not directly involved with the PMCs and HW events, instead it offloads the task of obtaining per-thread as well as system-wide PMC data to the monitoring module and then it uses a kernel API to retrieve the related obtained performance metrics from it.

In writing a monitoring module, the developer does not have to handle the low-level configuration and access to the counters. PMCTrack provides the programmer with an architecture-independent interface to easily configure the PMC units and select the desired events to be monitored in per-thread or system-wide mode. Additionally, it also enables to export the results of the requested metrics to user land, as *virtual counters* [cit need].

Apart from its OS oriented features, PMCTrack is also equipped with the user space library *libpmctrack*, which offers similar functionalities to PAPI, and some command-line tools meant to quickly obtain PMC data for selected user processes as well as assist the developer by providing useful information for the development of monitoring modules, e.g. available counters and events. The PMCTrack architecture is schematically depicted in figure 2.5.

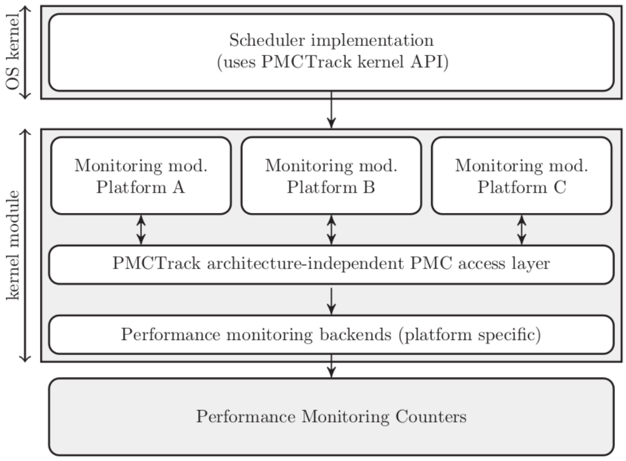

OS kernel
Scheduler implementation
(uses PMCTrack kernel API)
kernel module
Monitoring mod.
Platform A
Monitoring mod.
Platform B
Monitoring mod.
Platform C
PMCTrack architecture-independent PMC access layer
Performance monitoring backends (platform specific)
Performance Monitoring Counters

FIGURE 2.4: Interaction between OS scheduler and PMCTrack monitoring modules.

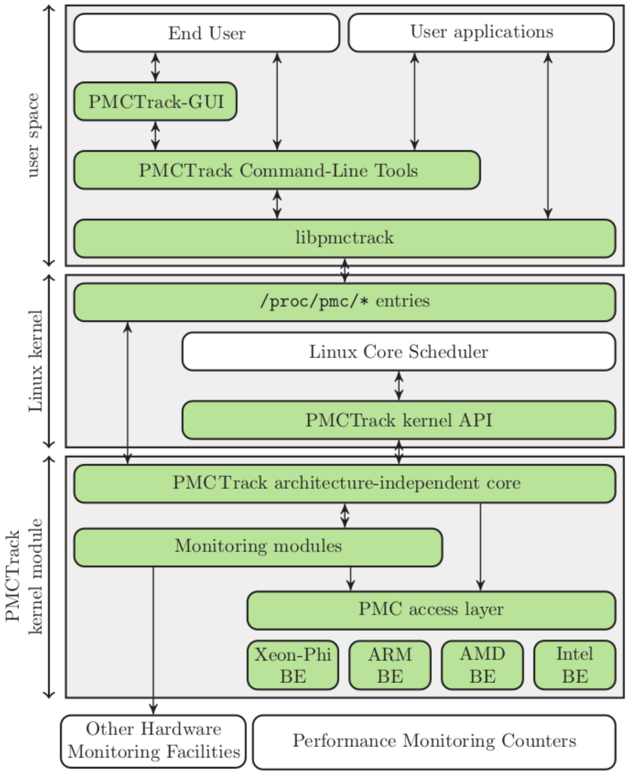


FIGURE 2.5: PMCTrack architecture.

Monitoring modules

As mentioned, monitoring modules are the platform-dependent units responsible of gathering PMC data and let them available to the scheduler. All the supported monitoring modules for a given platform are listed in the /proc/pmc/mm_manager file, that can be used in order to activate a specific monitoring module by writing to it. Only one monitoring module can be enabled at a time. The "pmc" directory is automatically created in the /proc filesystem when a PMCTrack module is inserted; it also includes, among others, the file /proc/pmc/config, which is used to specify configurations for a monitoring module, such as counters sampling period and size of the buffer where the values are stored.

The OS scheduler communicates with the active monitoring module in two steps: first, through the PMCTrack kernel API. The PMCTrack kernel module receives the notifications from the scheduler through the pmc_ops_t interface implemented in the PMCTrack architecture-independent core layer (fig. 2.5):

```
typedef struct __pmc_ops {
  /* invoked when a new thread is created */
  void* (*pmcs_alloc_per_thread_data)(unsigned long, struct
   task_struct*);
  /* invoked when a thread leaves the CPU */
  void (*pmcs_save_callback)(void*, int);
  /* invoked when a thread enters the CPU */
  void (*pmcs_restore_callback)(void*, int);
  /* invoked every clock tick on a per-thread basis */
  void (*pmcs_tbs_tick)(void*, int);
  /* invoked when a process invokes exec() */
  void (*pmcs_exec_thread)(struct task_struct*);
  /* invoked when a thread exits the system */
  void (*pmcs_exit_thread)(struct task_struct*);
  /* invoked when a thread descriptor is freed up */
  void(*pmcs_free_per_thread_data)(struct task_struct*);
  /* invoked when the scheduler requests per-thread
  monitoring information */
  int (*pmcs_get_current_metric_value)(
  struct task_struct* task, int key, uint64_t* value);
} pmc_ops_t;
```

The second step from the architecture-independent core layer to the active monitoring module is through the monitoring_module_t interface. Writing a monitoring module to be used from the scheduler entails implementing this interface, which turns out to be very similar to pmc_ops_t. It is essentially comprised of a set of callback functions that are triggered after a scheduling-relevant event occurs, e.g. module activation/deactivation, fork/exit syscalls, context switches, migrations, etc. The implementation of at least the necessary callback functions in the monitoring_module_t interface is required in order to enable a scheduling policy to use specific metrics based on PMCs data.

The scheduler can retrieve per-thread data of a given performance metrics computed by the monitoring module from the PMCTrack kernel API, in particular by calling the function pmcs_get_current_metric_value():

```
int pmcs_get_current_metric_value(struct task_struct* task,
                                  int key, uint64_t* value);
```

This function will in turn trigger the corresponding callback function implemented in the monitoring_module_t interface.

Chapter 3

Implementation

After having provided a wide overview of how Linux Completely Fair Scheduler (CFS) algorithm works and described Performance Counters (PMCs) and related SW tools needed to interact with them and retrieve their values, it is now time to dive into the core development of this thesis project. This can be summed up as follows:

- Development of a PMCTrack-compatible monitoring kernel module, which collects the desired PMCs data and metrics from kernel space
- CFS source code modifications, using the data collected by the above module

The kernel version considered in this project is 4.14.69, running on a Lenovo Ideapad 500S laptop.

3.1 Rationale

The main aim of this thesis is to enable the current default Linux scheduler CFS, to retrieve PMCs data dynamically collected at run-time and use them to alter its scheduling decision making. Specifically, this project focuses on monitoring the Last-Level Cache memory by using PMCs values to extract a meaningful metric of the dynamic cache behaviour of each process out of them.

3.1.1 CPU and I/O bound tasks in CFS

As explained in chapter 1, CFS algorithm relies on the concept of fairness among tasks: as seen, this is reached by assigning a fair portion of execution time to each runnable task, and then weighing this time-varying slice based on the *nice* value of the process, in order to reflect its original static priority and take it into account in the scheduling decision-making.

In doing so, CFS implicitly advantages I/O-bound processes with respect to CPU-bound ones, thanks to the combination of two of its key characteristics: one is the parameter used to sort tasks, i.e. the *virtual runtime*, and the other is the time allotted to each task for running on a CPU, i.e. the *dynamic timeslice*.

On one hand, `vruntime` represents, regardless of the "nice" correction, the effective measure of how long a task has already run on a CPU; the smaller its value, the sooner the process will have a chance to run again. At the same time, the timeslice given to each task for running without being preempted is computed as a predefined period in which all the runnable processes are guaranteed to be scheduled at least once, i.e. the *target latency*, divided by the number of runnable tasks in the queue, with no distinction among the types of processes involved. Therefore, both I/O-bound and CPU-bound processes will be provided with approximately the same amount of available processor time at each period; the formers will spend most of it waiting rather than running, while the latters will run more frequently and they will tend to use more CPU time than what they were given. As a result, I/O-bound tasks will expose a smaller `vruntime` and, as soon as a CPU-bound task expires its timeslice and the scheduler is invoked to check whether it should keep running or be preempted, it will be most likely preempted in favour of an I/O-bound process, because of its small virtual runtime. This is how CFS accomplishes great interactivity performances.

As mentioned in chapter 1, typical CPU-bound tasks, such as batch processes, belong from kernel perspective to the `SCHED_BATCH` policy, whose related scheduling entities are meant to be handled by CFS. As a consequence, despite this algorithm offering an effective and reasonable behaviour for desktop machines, it does still make sense to investigate how to enable the scheduler to better consider CPU-bound processes, especially in higher system load scenarios, that is when several CPU-bound runnable processes are contending for the same CPU.

3.1.2 Monitoring cache behaviour

When dealing with CPU-bound processes, their cache behaviour turns out to be quite relevant from a performance standpoint. The cache memory is a small and fast memory close to the processor which is used to store temporary data and instructions. It is usually divided into different *levels*, named as L1, L2, L3, etc. in such a way that small and fast caches get checked before the slower ones. It is common that the first cache levels are exclusive with respect to each core while the last-level cache (LLC) is shared among all CPUs.

In general, caching memory provides the processor with faster access to data that are frequently used during a program execution. Whenever the processor needs a data, it first looks into the cache. If the data is there, it will access it very fast; conversely, if it does not, it will read the data from the main memory and eventually copy it into the cache. These two cases are called *cache-hit* and *cache-miss*, respectively. In the event of a *cache-miss*, the data fetched from the main memory is copied together with the content of some of its adjacent memory locations, based on a *continuity* principle: if, at a certain time instant t, the CPU needs a data stored at location x, as it advances the program execution, at a future instant $(t+1)$, it will probably need data stored at address $(x+1)$.

Thanks to the exponential growth of the gap between processor and memory performances in favour of the first, a good cache behaviour plays an important role when addressing the performances of recent computer systems. Reducing the average access time of data from the main memory can also lead to reduce the average power consumption[cit]. Improving cache behaviour for a process entails minimising the number of cache-misses occurred during its execution, thus keeping the cache *hot*. PMCs can provide such insights, by configuring them to counts the occurrences of specific cache events. However, the main drawback of this approach boils down to the fact that PMCs data can sometimes include speculative information. This is true when, for instance, cache-misses counted on the LLC are affected by hardware prefetches that misses a previous cache level.[cit]

3.1.3 How PMCs enter the CFS

In the project carried out in this thesis, the attention is focused on enabling the Linux scheduler to retrieve cache-related PMCs data and use them to alter the standard scheduling algorithm based on those information. More specifically, our metric of interest is the cache-miss rate of the LLC, and PMCs have been programmed and used to extract such metric. This is usually the largest cache memory in the system and the only one being shared among all cores.

Regardless the implementation details of how PMCs data are actually collected and how they enter the CFS source code, that will be further described in depth, it is worth discussing the reasoning and the logic behind them.

The CFS key variable that is examined and altered by considering the cache-miss rate is the virtual runtime of each task in the system. As mentioned, this variable affects the position in which the runnable entities, including actual tasks, are sorted in the *rbtree* of each CPU's runqueue, thus determining the order in which the they will be picked up for running on that core.

In particular, a correction on the variation of virtual runtime, that is periodically computed and used to update the variable for each process, has been introduced, taking into consideration the task's cache-miss rate and its weight with respect to the other processes in the same runqueue. If $cm(t)$ denotes the number of cache-misses at a certain instant t, it is possible to define the cache-miss rate of a process p as:

$$r_p(t) = \frac{dcm(t)}{dt} \tag{3.1}$$

This variable is effectively computed in the kernel as the number of cache misses revealed divided by the number of clock cycles counted, multiplied by a correcting factor.

$$r_p \approx \frac{N_{cm}}{N_{cyc}} \cdot K \tag{3.2}$$

The correcting factor K is introduced in order to resize the quantity and obtain meaningful numbers. Since floating point numbers require special support to be used inside kernel code, and this is also typically avoided, K is simply introduced to pull out rates greater than 1.

When the kernel updates the virtual runtime for each task, it normally computes a variation of it that is added to the previous value. This *delta* is simply the elapsed execution time weighted based on the "niceness" of the process. By calling this variation Δ_{VR}, the new corrected variation of virtual runtime is computed as:

$$\Delta_{VR}^* = \left(1 + \frac{\Delta r_p(T)}{\sum_{i=0}^{n-1} |\Delta r_i(T)| + |\Delta r_p(T)|}\right) \cdot \Delta_{VR} \tag{3.3}$$

where

- $\Delta r(T) = r(T) - r(T-1)$
- $T :=$ current time sample in which the rate value is extracted
- $p :=$ current process being updated
- $i :=$ generic process in the same runqueue of p

The ratio in equation 3.3 represents the cache-miss rate variation for the current process p with respect to the other runnable tasks in the same runqueue. It is quite clear from the formula that it can assume numerical values in the range $[-1, 1]$. In particular, letting R be this ratio, and remembering that a lower value of `vruntime` for a task implies more chances to be picked up for running, it is worth analysing the following cases:

- $0 < R < 1$: the cache-miss rate of p has increased and there are other runnable processes in the runqueue; a penalty is imposed by raising up Δ_{VR} of $R\%$
- $-1 < R < 0$: the cache-miss rate of p has decreased and there are other runnable processes in the runqueue; a reward is imposed by lowering Δ_{VR} of $R\%$
- $R = 0$: Δ_{VR} is unaffected
- $R = 1$: the cache-miss rate of p has increased but there are no other processes in the runqueue; Δ_{VR} is unaffected
- $R = -1$: the cache-miss rate of p has decreased but there are no other processes in the runqueue; Δ_{VR} is unaffected

In this way, at each sampling period in which the active monitoring module reads the PMCs, the virtual runtime of each task in the system is adjusted, based on its measured cache-miss rate variation with respect to the rate variations of the other runnable tasks in the runqueue.

In the next sections both the implementation details of the monitoring kernel module and the scheduler modifications performed will be presented and discussed in details.

3.2 PMCTrack monitoring module

In order to gather PMCs data from kernel space and enable the scheduler to use these data to create performance metrics, PMCTrack tool has been used. With respect to other tools, that most of the time allow to monitor PMCs only in user space, this tool is definitely OS-oriented, and therefore has been adopted.

At this purpose, the PMCTrack kernel monitoring module `llc_monitoring_mm.c` has been developed. As described in chapter 2, such a module is constituted of some global configuration variables and a set of callbacks, automatically called at each occurrence of some scheduling events. The set of callback functions are then used to create the `monitoring_module_t` interface needed by PMCTrack tool to operate.

The module is compatible with Intel architectural performance monitoring and it is instrumented to monitor the following two events per each task:

- Last Level Cache Misses
- Unhalted Core Cycles

This pair of events is imposed by declaring a global constant configuration string:

```c
#define LLC_MODEL_STRING "LLC cache-misses monitoring module"
/* Global PMC config for each task in the system */
const char *llc_monitoring_pmcstr_cfg[] = {
  "pmc3=0x2e,umask3=0x41,pmc1", NULL
  };
```

As shown in the above snippet of code, the raw counters configuration string is composed by three terms in comma-separated-value format: the first two `pmc3` and `umask3` refer to the "Event select" and "UMask" fields to count the LLC misses, respectively, while the third member is used to pull out the Unhalted Core Cycles by using the fixed-function counter pmc1.

The module implements all the callbacks that constitute the monitoring_module_t interface, which interacts with the PMCTrack kernel API:

```
/* Implementation of the monitoring_module_t interface */
monitoring_module_t llc_monitoring_mm = {
  .info = LLC_MODEL_STRING,
  .id = -1,
  .enable_module = llc_monitoring_enable_module,
  .disable_module = llc_monitoring_disable_module,
  .on_fork = llc_monitoring_on_fork,
  .on_free_task = llc_monitoring_on_free_task,
  .on_new_sample = llc_monitoring_on_new_sample,
  .get_current_metric_value =
   llc_monitoring_get_current_metric_value,
  .module_counter_usage = llc_monitoring_module_counter_usage
};
```

Neglecting the trivial .info and .id fields of the interface, an exhaustive description of each function is provided in the next sections.

The module interface with the PMCTrack kernel API is also composed by another component, that is a per-task private data-structure. In llc_monitoring_mm.c, this structure is the llc_monitoring_thread_data_t:

```
/* Per-task private data */
typedef struct {
  metric_experiment_t llc_metric;
  unsigned int samples_cnt;    // number of samples
  uint64_t cur_llc_miss_rate; // LLC miss/cycles
  uint64_t cur_llc_misses;  // LLC miss count
  char is_new_sample;      // ready sample flag
}llc_monitoring_thread_data_t;
```

This data structure contains a per-task PMCTrack metric, its current outputs, i.e. LLC misses rate and raw misses counts, as well as the number of samples collected and a flag used to specify if a new sample is available.

3.2.1 Module probe

The first function of the interface implemented by the module is the function llc_monitoring_enable_module(), which is triggered when the module itself is loaded into the system by means of the modprobe or the insmod Linux commands:

```
static int llc_monitoring_enable_module(void)
{
  if (configure_performance_counters_set(
            llc_monitoring_pmcstr_cfg,
            llc_monitoring_pmc_configuration, 1)) {
    printk("Cannot configure global performance counters...\n");
    return -EINVAL;
  }
  init_llc_metric(&llc_monitoring_metric_exp);
  printk(KERN_ALERT "%s has been loaded successfuly\n",
         LLC_MODEL_STRING);

  return 0;
}
```

The above function calls the PMCTrack internal routine configure_performance_counters_set(), which is in charge of copying the global configuration string previously declared into an array of global configurations (unitary, in this case), i.e. the *experiment set array*. This variable is of type core_experiment_set_t (internal PMCTrack data type) and it is also declared in the module:

```
core_experiment_set_t llc_monitoring_pmc_configuration[1];
```

This step can appear a bit confusing at a first sight. Actually it allows to simplify the configuration phase, since the experiment set acts as a global container for all counters configurations considered by the module. This is basically aimed at avoiding making a copy of the configuration string each time that a new task is forked from a parent. Since the objective of the monitoring module is to monitor the same set of HW events, i.e. LLC misses and cycles, for all the tasks belonging to CFS scheduling class, using a unitary experiment set containing the global configuration string allows to configure the counters only once when the module is probed into the system, rather than each time that a process is forked.

Then the callback calls in turn the helper function init_llc_metric() that is used to setup and initialise the output metric computed by the module. Specifically, this function instructs the module to combine the values of the two monitored events for each task by taking their ratio, in order to extract a cache-miss rate metrics. Similarly to the counter configuration phase, a module global variable is also needed to be used as container for the output metric settings. This time the variable belongs to the internal metric_experiment_set_t type:

```
metric_experiment_t llc_monitoring_metric_exp;
```

This variable is passed as argument to init_llc_metric(), whose definition is:

```
static void init_llc_metric(metric_experiment_t *metric_exp)
{
  pmc_metric_t *cmr_metric = NULL;
  pmc_metric_t *cm_metric = NULL;
  init_metric_experiment_t(metric_exp, 0);
  cmr_metric = &(metric_exp->metrics[0]);
  cm_metric = &(metric_exp->metrics[1]);
  metric_exp->size = 2; //2 HW Counters involved
  pmc_arg_t arguments[2];
  arguments[0].index = 0;
  arguments[0].type = hw_event_arg;
  arguments[1].index = 1;
  arguments[1].type = hw_event_arg;
  init_pmc_metric(cmr_metric, "LLC_miss_rate", op_rate,
                  arguments, 100);
  init_pmc_metric(cm_metric, "LLC_miss_count", op_none,
                  arguments, 1);
}
```

As shown, the input arguments of the LLC miss rate metric, referenced by cmr_metric, are the two HW events counts, and the metric is initialised by init_pmc_metric(). It is worth noticing that these arguments are combined in the same order as they appear in the global configuration string, i.e. first and second arguments are "LLC misses count" and "cycles count", respectively. Furthermore, the division operation is imposed by passing the third argument op_rate while the last input (100) is a correcting multiplicative factor, introduced to avoid integer truncation when computing the rate.

Actually, apart from the rate, the raw misses count for each process is also kept by the module into another metric, i.e. cm_metric.

3.2.2 Task fork

Whenever a new forked child process enters in the system, the callback function llc_monitoring_on_fork() is invoked, in order to impose the desired PMCs configuration and initialise the current values of the output metric. The function appears as:

```
static int llc_monitoring_on_fork(unsigned long clone_flags,
                                  pmon_prof_t *prof)
{
  llc_monitoring_thread_data_t *data = NULL;

  if (prof->monitoring_mod_priv_data != NULL)
    return 0;

  /* Clone global experiment set, i.e. counters configuration,
   in child process private data */
  if (!prof->pmcs_config) {
    clone_core_experiment_set_t(&prof->pmcs_multiplex_cfg[0],
                           llc_monitoring_pmc_configuration);
    /* Update current experiment set, i.e. counters
   configuration, of child process */
    prof->pmcs_config =
    get_cur_experiment_in_set(&prof->pmcs_multiplex_cfg[0]);
  }

  data = kmalloc(sizeof(llc_monitoring_thread_data_t),
                 GFP_KERNEL);
  if (data == NULL)
    return -ENOMEM;

  memcpy(&data->llc_metric, &llc_monitoring_metric_exp,
         sizeof(metric_experiment_t));

  data->samples_cnt = 0;
  data->cur_llc_misses = 1;
  data->cur_llc_miss_rate = 1;
  data->is_new_sample = 0;
  prof->monitoring_mod_priv_data = data;

  if (!(dl_prio(prof->this_tsk->prio) ||
      rt_prio(prof->this_tsk->prio)))
    prof->this_tsk->prof_enabled = 1;

  return 0;
}
```

Regardless of the clone_flags, this callback, as well as many of the following ones, takes a pointer to a special pmon_prof_t data structure. This data structure is where the kernel module stores monitoring information for a thread[cit need]. A pointer to this structure is stored in the prof field added to the task descriptor by the PMCTrack kernel patch.

The above routine performs the following operations:

- Clone the global experiment set into the child's private data and select the counters configuration (only one in this case)
- Allocate memory for the module's per-task private structure, i.e. `llc_monitoring_thread_data_t`
- Copy the global PMC metric into the private module's per-task private structure
- Initialise the values of current metrics
- Dereference pointer to the private monitoring per-task data to update it with the current module's monitoring private data
- Enable monitoring the current task only if it belongs to the CFS scheduling class; this is done by checking that the current task has no other scheduling policy related priority

The profiling mode for the newly forked task is activated by setting the `prof_enabled` flag, that is embedded into the task descriptor when applying the PMCTrack kernel patch.

3.2.3 New sample collected

The bulk of the functionality implemented by `llc_monitoring_mm.c` is to periodically read the desired PMCs values and extract a meaningful metric out of them, to make it available from the scheduler code. This job is handled by the function `llc_monitoring_on_new_sample()`:

```c
static int llc_monitoring_on_new_sample(pmon_prof_t *prof,
     int cpu, pmc_sample_t *sample, int flags, void *data)
{
  llc_monitoring_thread_data_t *llc_data;
  llc_data = prof->monitoring_mod_priv_data;

  if (llc_data != NULL) {
    metric_experiment_t *metric_exp = &llc_data->llc_metric;
    compute_performance_metrics(sample->pmc_counts, metric_exp);
    llc_data->cur_llc_miss_rate = metric_exp->metrics[0].count;
    llc_data->cur_llc_misses = metric_exp->metrics[1].count;
    llc_data->samples_cnt++;
    llc_data->is_new_sample = 1;
  }
  return 0;
}
```

Each time that a new sample is collected, the above function is invoked and calls in turn `compute_performance_metrics()` in order to update the output metric in the module's private per-task structure. After that, the current cache-miss rate, as well as the misses counts, are updated in the same private structure. Finally the `is_new_sample` flag is set to notify that the a new sample has been just collected. This will be used directly by the scheduler code when periodically retrieving the output rate value.

3.2.4 Get current metric value

As already discussed at the end of chapter 2, the scheduler can communicate with the active monitoring module to retrieve its output metric data by calling the following function from the PMCTrack kernel API:

```
int pmcs_get_current_metric_value(struct task_struct* task,
                                  int key, uint64_t* value);
```

This kernel API function has of course its module's counterpart that is eventually wrapped by the above function, to perform the actual job. This module's internal routine is `llc_monitoring_get_current_metric_value()`:

```
static int llc_monitoring_get_current_metric_value(
           pmon_prof_t *prof, int key, uint64_t *value)
{
  llc_monitoring_thread_data_t *llc_data;
  llc_data = prof->monitoring_mod_priv_data;

  if (llc_data == NULL)
    return -1;

  if (key == CACHE_MISS_RATE && llc_data->is_new_sample) {
    (*value) = llc_data->cur_llc_miss_rate;
    llc_data->is_new_sample = 0;
    return 0;
  }
  else
    return -1;
}
```

When this function is invoked, it first checks, as all the previous function do, that the private monitoring per-task data structure contains data, otherwise it has to return immediately. Then, the `key` argument, which is just an internal PMCTrack identifier to the each metric possibly computed by all the compatible monitoring modules, is also checked together with the `is_new_sample` flag. Ultimately, if the desired metric is the cache-miss rate, and a new value for it is effectively available, the current cache-miss rate value is copied from the module's private per-task data structure into the variable referenced by the last argument `value`, thus making it available to the scheduler.

3.2.5 Free task

When a task exits the system, the kernel has to release the resources allocated for that task. Among them, if a monitoring module is active, it needs to deallocate also its monitoring private data and this is done by `llc_monitoring_on_free_task()`:

```
static void llc_monitoring_on_free_task(pmon_prof_t *prof)
{
  llc_monitoring_thread_data_t* llc_data;
  data = prof->monitoring_mod_priv_data;
  if (llc_data)
    kfree(llc_data);
}
```

The above routine simply uses `kfree` to deallocate the monitoring private data for the exiting process.

3.2.6 Module unprobe

The last function implemented in llc_monitoring_mm.c is the callback triggered when the module itself is disabled by means of the modprobe or rmmod Linux commands.

```
static void llc_monitoring_disable_module(void)
{
  free_experiment_set(&llc_monitoring_pmc_configuration[0]);
  printk(KERN_ALERT "%s monitoring module unloaded!!\n",
         LLC_MODEL_STRING);
}
```

This function just releases the resources related to the experiment set previously created to impose the desired PMCs configuration by invoking free_experiment_set(). Then, it notifies that by sending a message visible on the main kernel logging console.

3.3 Main scheduler modifications

It is now time to discuss in details how the CFS algorithm makes use of the information collected by the PMCTrack monitoring module and how these data alter its standard behaviour. The conceptual logic behind such modifications has been already described and how those information enter the scheduler have already been described: Therefore, it is now possible to dive into the implementation details.

The first step in order to enable the kernel to use the monitoring module previously described, as well as to let the scheduler communicate with it, is to install the PMCTrack Linux kernel patch for kernel version 4.14.69. This patch introduces modifications and adds a few lines of code in some sections of the core scheduler code. Additionally, it also adds the pmctrack.c file, and its header, to the main Linux source tree. These files implement the PMCTrack kernel API that is used by the scheduler to interface with the active monitoring module.

3.3.1 New scheduler entries

In order to use PMCs data gathered by the monitoring module to implement the logic described in 3.1.3, some kernel variable needs to be stored by means of some additional entries in the main data structures involved with the CFS algorithm.

More specifically, the current cache-miss rate and the last computed difference with the previous rate are added to the sched_entity structure, that is in turn embedded in the process descriptor. The new data structure, defined in <include/linux/sched.h>, now appears as:

```
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;
  u64         prev_sum_exec_runtime;

  u64         nr_migrations;

  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

  #ifdef CONFIG_SMP
  /*
  * Per entity load average tracking.
  *
  * Put into separate cache line so it does not
  * collide with read-mostly values above.
  */
  struct sched_avg    avg ____cacheline_aligned_in_smp;
  #endif

+ #ifdef CONFIG_PMCTRACK
+ u64 cache_miss_rate;
+ s64 delta_cache_miss_rate;
+ #endif

};
```

The new added entries are `cache_miss_rate` and `delta_cache_miss_rate`. The latter is exactly the numerator of the ratio in the formula 3.3.

Furthermore, the two new entries are both initialised to 0 in the helper function `__sched_fork()` in <kernel/sched/core.c>:

```
static void __sched_fork(unsigned long clone_flags, struct
    task_struct *p)
{
  p->on_rq      = 0;

  p->se.on_rq     = 0;
  p->se.exec_start    = 0;
  p->se.sum_exec_runtime    = 0;
  p->se.prev_sum_exec_runtime = 0;
  p->se.nr_migrations   = 0;
  p->se.vruntime      = 0;

#ifdef CONFIG_PMCTRACK
  p->se.cache_miss_rate = 0;
  p->se.delta_cache_miss_rate = 0;
#endif

  ...
```

Besides, also the sum of the cache-miss rate variations for tasks belonging to the same CFS runqueue needs to be stored somewhere, in order to be available to the kernel. This entry is the `runnable_delta_cm_rate` and it is included in the CFS runqueue's definition in <kernel/sched/sched.h>:

```c
struct cfs_rq {
  struct load_weight load;
  unsigned int nr_running, h_nr_running;

  u64 exec_clock;
  u64 min_vruntime;
#ifndef CONFIG_64BIT
  u64 min_vruntime_copy;
#endif

  struct rb_root_cached tasks_timeline;

  /*
  * 'curr' points to currently running entity on this cfs_rq.
  * It is set to NULL otherwise (i.e when none are currently running).
  */
  struct sched_entity *curr, *next, *last, *skip;

  ...

#ifdef CONFIG_PMCTRACK
  u64 runnable_delta_cm_rate;
#endif
  };
```

This variable, instead, constitutes the denominator of the ratio in equation 3.3. It is also initialised to 0 inside the function `init_cfs_rq()`, defined in <kernel/sched/fair.c>:

```c
void init_cfs_rq(struct cfs_rq *cfs_rq)
{
  cfs_rq->tasks_timeline = RB_ROOT_CACHED;
  cfs_rq->min_vruntime = (u64)(-(1LL << 20));

  ...

#ifdef CONFIG_PMCTRACK
  cfs_rq->runnable_delta_cm_rate = 0;
#endif
}
```

In conclusion, the three entries added, i.e. the per-process cache-miss rate as well as its variation and the per-queue sum of cache-rate miss variations, belong to `u64` and `s64` data types. These are the kernel opaque types standing for the more common `uint64` and `int64`, which all entail 8 bytes of occupied memory each. Therefore, the total amount of memory added by the modified algorithm with respect to the PMCTrack kernel patch is 24 bytes.

3.3.2 Modify the virtual runtime

Once the scheduler is geared to handle the new introduced quantities, it is possible to modify the code sections where the `vruntime` is updated. As widely described in chapter 1, this happens in the function `update_curr()` in <kernel/sched/fair.c>, where the scheduling variables of interest are updated for the currently running entity:

```c
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;
  u64 delta_fair;

#ifdef CONFIG_PMCTRACK
  u64 cm_rate;
  s64 del_cm_rate;
  u64 del_cm_rate_sum;
  struct sched_entity *pos, *n;
#endif

  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);

  delta_fair = calc_delta_fair(delta_exec, curr);

#ifdef CONFIG_PMCTRACK
  if (entity_is_task(curr)) {
    if (task_of(curr)->prof_enabled &&
      pmcs_get_current_metric_value(task_of(curr),
                                    CACHE_MISS_RATE,
                                    &cm_rate) != -1) {
      if (cm_rate != 0) {
        trace_printk("(prev) CMR = %llu; del_CMR = %lld;
                     sum_del_CMR = %llu\n",
                     curr->cache_miss_rate,
                     curr->delta_cache_miss_rate,
                     cfs_rq->runnable_delta_cm_rate);
        trace_printk("delta = %llu\n", delta_fair);

        del_cm_rate = cm_rate-curr->cache_miss_rate;
        del_cm_rate_sum = abs(del_cm_rate);

        rbtree_postorder_for_each_entry_safe(pos, n,
                &cfs_rq->tasks_timeline.rb_root, run_node) {
          del_cm_rate_sum += abs(pos->delta_cache_miss_rate);
        }

        curr->cache_miss_rate = cm_rate;
        curr->delta_cache_miss_rate = del_cm_rate;
        cfs_rq->runnable_delta_cm_rate = del_cm_rate_sum;

        delta_fair = corr_delta_fair(delta_fair,
                              curr->delta_cache_miss_rate,
                              cfs_rq->runnable_delta_cm_rate);

        trace_printk("(post) CMR = %llu; del_CMR = %lld;
                     sum_del_CMR = %llu\n",
                     curr->cache_miss_rate,
                     curr->delta_cache_miss_rate,
                     cfs_rq->runnable_delta_cm_rate);
        trace_printk("corr_delta = %llu\n", delta_fair);
      }
    }
  }
  #endif

  curr->vruntime += delta_fair;
  update_min_vruntime(cfs_rq);

  ...

}
```

Basically, first the three local variables, i.e. cm_rate, del_cm_rate and del_cm_rate_sum are declared. Additionally, the variation of virtual runtime to be eventually added, is also declared as a local variable delta_fair. This is first computed by the function calc_delta_fair(), as the elapsed execution time delta_exec weighted based on the "nice" priority of the current process.

This is the point where the main scheduler behaviour is significantly altered. First, a check on what the current entity represents is performed: since scheduling entities can also identify group of processes or users, while our analysis focuses on actual tasks, the routine entity_is_task() is used to determine whether the considered entity is a pure task.

If that is the case, the next step is to verify that the profiling mode, operated by the active monitoring module, is enabled for the current task and, if yes, the scheduler needs to ask the monitoring module if a new sample has been collected. The first check is done on the prof_enabled flag of the task, which is set by the active monitoring module when a new task, whose policy is mapped to CFS, is forked.

The availability of a new metric's value is, instead, internally verified by the function pmcs_get_current_metric_value(), since it calls in turn its module's counterpart, i.e. llc_monitoring_get_current_metric_value(). The module's callback, in fact, performs a check on the is_new_sample flag and, if it is clear, it returns -1. Conversely, if a new sample is available, the last collected metric's value is copied into the variable cm_rate.

After that, the difference with respect to the previous rate for the current process is computed in del_cm_rate, whose absolute value is used to initialise del_cm_rate_sum and start building the denominator of the ratio in equation 3.3.

Then, the sum of the rates variations of the runnable tasks in the runqueue needs to be updated. This is performed by iterating over the current CFS runqueue, i.e. which is an *rbtree*, and adding the module of the rate's difference for each process in the tree. At this purpose, the internal kernel macro rbtree_postorder_for_each_entry_safe is invoked. As shown in the code, and according to the kernel implementation of red-black trees, two auxiliary pointers to sched_entity structures, i.e. pos and n, are passed as first two arguments; then, the macro needs to receive the address of the root node of the tree as well as the name of the rb_node field embedded in the `sched_entity` structure, which represents the link between the entity itself and the *rbtree* that contains it as a node. This field is exactly the `run_node` member.

Once the iteration is complete and the sum of the rates variations of the other runnable tasks is updated in `del_cm_rate sum`, it is possible to update all the variables in the `sched_entity` structure of the current process as well as in the `cfs_rq`.

Finally, it is possible to correct the variation of `vruntime` to be added to `curr->vruntime`. In order to do that, another function `corr_delta_fair()` has been implemented in the same source file:

```c
static u64 corr_delta_fair(u64 delta, s64 del_CMR,
                           u64 run_del_CMR)
{
  unsigned int d1, d2, d3;

  /* If delta_cache_miss_rate of p (num),
    runnable_delta_cm_rate of cfs_rq (den)
  are zero or if the rate of p is greater or equal than the
    runnable rate, nothing is to be done */
  if ((del_CMR == 0)||(run_del_CMR == 0)||(abs(del_CMR) >=
    run_del_CMR))
    return delta;

  /* Compute first three decimal digits of ratio */
  d1 = 10*abs(del_CMR)/run_del_CMR;
  d2 = 100*abs(del_CMR)/run_del_CMR-10*d1;
  d3 = 1000*abs(del_CMR)/run_del_CMR-100*d1-10*d2;

  /* Round to the second decimal digit */
  if (d3 >= 5) {
    d2++;
    if (d2 == 10) {
      d2 = 0;
      d1++;
      if (d1 == 10)
        d1 = 0;
    }
  }

  /* Based on the sign of the numerator, add or subtract the
    percentage */
  if (del_CMR > 0)
    delta = delta+(d1*delta)/10+(d2*delta)/100;
  else
    delta = delta-(d1*delta)/10-(d2*delta)/100;

  return delta;
}
```

It is worth spending some words on the above routine. This is the function which implements the logic discussed for equation 3.3. If either the numerator `del_CMR` or the denominator `run_del_CMR` is null, the function returns immediately and Δ_{VR}) is not altered at all. This behaviour is imposed also if the module of the numerator is greater (unlikely) or equal to the denominator; this second case occurs, as described in section 3.1.3, when no other runnable task fills in the *rbtree* of the currently running process ($R = \pm 1$). In such scenario, it does not make any sense to introduce the correction.

If, conversely, other runnable processes populate the tree, the function needs to compute the new vruntime variation $\Delta^*_{VR})$. With reference to section 3.1.3, the percentage on the original variation to be added or subtracted is exactly the ratio R between del_CMR and run_del_CMR, which results in a numerical quantity lying in the range $(-1;1)$. Unfortunately, the usage of floating point numbers inside the kernel requires some special support and, regardless of the general reluctance of kernel developers in doing so, this is particularly unfeasible in critical code paths, such as inside the scheduler.

Therefore, another approach has been adopted, in order to properly handle the computation of Δ^*_{VR}. The three local variables d1, d2 and d3 are introduced to compute the tenth, hundredth and thousandth part of R, respectively, into an integer fashion. Therefore, after properly rounding the quantities to the second decimal digit, the new vruntime variation can be computed by simply adding or subtracting, depending on the sign of the numerator, the following term:

(d1*delta)/10 + (d2*delta)/100

In this way, the floating point calculations are avoided and converted into an integer sum, or difference, based on the sign of the cache-miss rate variation of the current task. In conclusion, the adjusted delta_fair can now finally be added to curr->vruntime.

In the next and final chapter of this thesis, the test environment for the modified scheduler will be described and the results will be presented and discussed.

Chapter 4

Test environment

In this last chapter, a suitable testing environment used to pull out results and comparisons between the standard Linux kernel 4.14.69 and the proposed scheduler patch with dynamic PMCs monitoring, will be presented.

In particular, a short overview of the selected programs included from the "MiBench" test suite will be first provided, in order to introduce how the selected applications have been employed to build two singular multithreaded programs. The aim of these two testing software concerns the simulation of different scenarios of multiple processes coexistence, thus characterising and benchmarking the scheduler behaviour obtained by running them on the proposed CFS variation with cache-miss rate dynamic monitoring.

Finally, the retrieved results and graphs will be widely presented and discussed.

4.1 MiBench test suite

In order to create a multithreading scenario for testing the scheduler, a sub-set of the commercially representative benchmark test suite MiBench [12] has been considered. Specifically, the automotive subset of benchmark programs has been picked up and wrapped into two multiprocess launcher executables, aimed at creating two different running conditions for the same set of automotive programs. The launchers perspective and the related simulations designed will be further discussed in the next section.

The automotive category among MiBench benchmark consists in a set of basic programs, performing mathematical operations, bit manipulation, data input-output and organisation. This group of programs includes a basic math test, a bit counting test, a sorting algorithm and a shape recognition application. It is composed by the following main programs:

- *basicmath*: simple math test performing basic mathematical computations. The program includes calculations such as cubic functions, integer square root, angles format conversions and takes a small and large set of constants as input

- *bitcount*: bit counting algorithm implementing five distinct methods to count the number of bits in an input integer array of 1's and 0's. The number of iterations to perform is specified as executable input

- *qsort*: sorting program based on quick sort algorithm that rearranges an array of strings in ascending order. It considers both a small list of words and a large three-dimensional data set of points

- *susan*: image recognition suite implementing smoothing and adjusting operations on an input image. The small input data is a simple black and white rectangle, while a complex picture is used as large input. It can run in the following three modes:

a. *smoothing mode*
b. *edges mode*
c. *corners mode*

4.2 Multithreading test application

From a scheduler perspective, the above applicative programs in run can be considered as a set of CPU-bound processes. In order to consistently test and stress the scheduler, aimed at observing the modified scheduler's behaviour with respect to the standard version, it is necessary to create an interleaving of multiple processes together.

At this purpose, the applications described in section 4.1 have been wrapped together into two multithreaded launcher applications, whose function is to fork several child tasks from the main thread, each associated to one particular program, in two different fashions. In doing so, each of the six programs results in, at least, two instances of itself obtained by feeding the same application with both small and large input data, thus giving rise to a total testing framework of twelve processes.

Two different multiprocess testing scenarios have been conceived and designed, resulting in the implementation of two multiprocess launchers, based on the same set of twelve processes:

- *MiBenchSingleParallelLauncher*
- *MiBenchMultiParallelLauncher*

Both implemented tests employ a wrapper function that is used to execute a specific program's code in each child process created. A general feature of both launchers is the randomisation of the children creation, which is differently achieved in the two structural variants that will be discussed in the next sections. This has been done in order to avoid a fixed processing order of operations performed by the children tasks at each launch iteration.

The ultimate goal of the two test programs is to retrieve the execution times for the whole set of processes considered. In particular, all tasks data are sorted based on the program source that is effectively executed in each child. The mean value and the standard deviation over 50 launches of each benchmark program are computed in units of *milliseconds* by both tests.

4.2.1 Single-parallel execution test

The first test program implemented is a *single-parallel* launcher: it is *single-parallel* in the sense that all child processes are forked in groups of 12, with one instance for each type of benchmark program considered. Then, this pattern is iterated 50 times and the order by which the 12 children are generated is randomised at each iteration, in order to have a consistent analysis from a statistical point of view. A graphical scheme of the implemented test routine is shown in 4.1.

As seen, the parent main launcher process starts first. After generating a random sequence for the children processes creation, it advances its execution by `fork`ing 12 tasks, each corresponding to a different benchmark program. In each child code, a

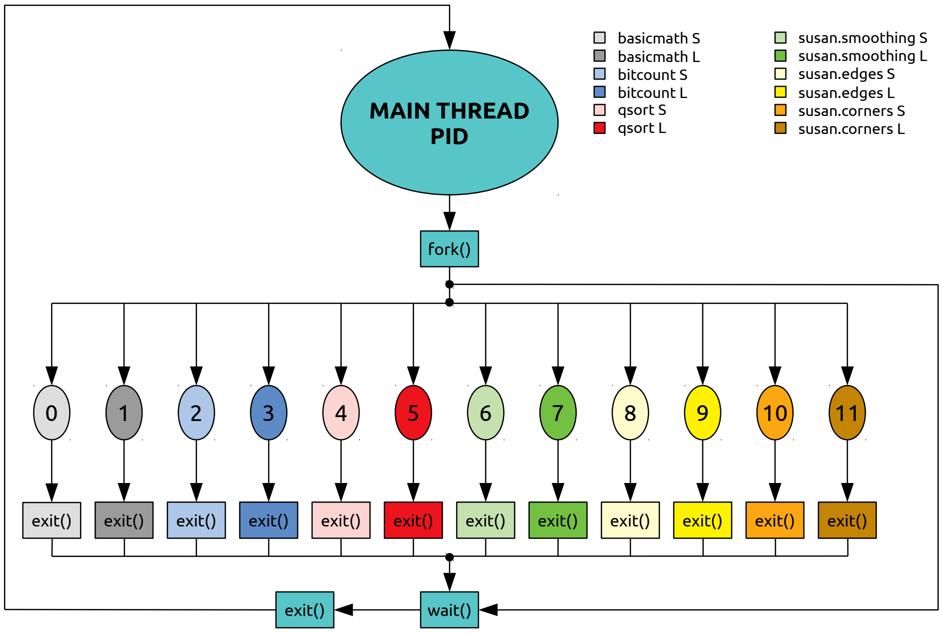


FIGURE 4.1: MiBench-based single-parallel test scheme.

wrapper is invoked in order to separately execute the program's source code corresponding to that child. The wrapper function also profiles the execution time of the process and stores its *PID* and the elapsed time to a specific memory area, that is shared between the parent and its children. After each execution, the child task calls an `exit()` syscall in order to terminate its execution. In this test, the parent thread `wait`s for all children completion, before resuming itself and copying the content of the shared memory section containing all the children's execution times and PIDs for the current iteration in its address space. This processes launch pattern is iteratively performed 50 times.

Finally, after having copied the content of all shared memory segments of each iteration into its address space, the min thread can compute mean values and standard deviation of its past children tasks, that are grouped based on the benchmark program associated with each child.

4.2.2 Multi-parallel execution test

The second test program is a *multi-parallel* launcher, slightly different with respect to its single counterpart, as far as the launch of the children tasks is concerned. The same set of 12 processes is still launched 50 times but, this time, all instances of a same benchmark program are let run together, thus subjecting the scheduler to a larger stress, due to the greater number of tasks populating the system. The children generation order is still randomised, resulting in only one launch of 600 processes, each associated with one benchmark program, with 50 instances of the same benchmark application. A graphical scheme of this test routine can be seen in figure 4.2.

Also in this new configuration, the parent main launcher process starts its execution. As before, it first generates the random sequence used to create the children. This time the series must be applied to 600 tasks, equally distributed among the 12

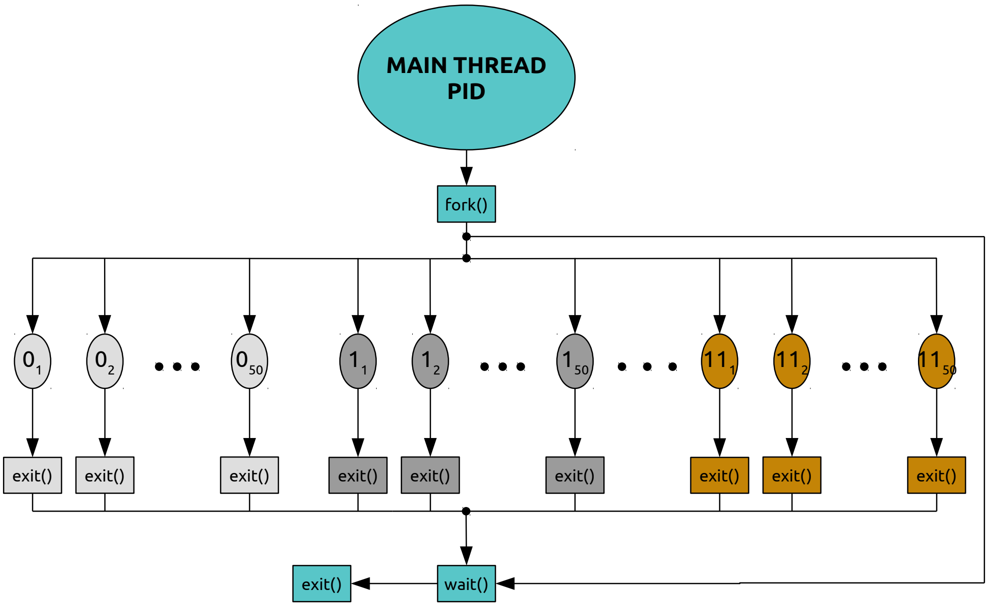


FIGURE 4.2: MiBench-based multi-parallel test scheme.

benchmark programs considered. Then, the parent can `fork` all its children tasks, according to the previously computed random pattern. In each child code, the same wrapper function already used in the single-parallel test is invoked, in order to let each child execute a specific program code. As before, while the children are running, the parent thread `waits` for them. A larger shared memory section is needed to let the children write their PIDs and profiled execution times to it, thus allowing the parent to retrieve these data after all children complete. Once each task has executed and stored its desired information in the shared memory, it can safely cease its execution by calling `exit()`. Once all the children have completed, the parent can attach the shared memory to its address space and gather all the data of its children and, as before, compute mean values and standard deviations for all the tasks associated with the same benchmark source code.

4.3 Experimental setup

Once the two multithreading applications *MiBenchSingleParallelLauncher* and *MiBenchMultiParallelLauncher*, designed to test the scheduler, have been described, it is time to illustrate the testing conditions of the experiments and finally comment the obtained results.

First, the machine on which the tests have been executed is a Lenovo Ideapad 500S-13ISK, equipped with an Intel Core i5-6200U processor; this is a symmetrical quad-core microprocessor, with each core running at a maximum clock speed of 2.30 GHz. Linux kernel version 4.14.69 is running on the considered machine.

The experiments have been carried out on three different versions of the above kernel:

- *CFS baseline*: a fresh and clean Linux 4.14.69 installation

- *CFS cache-miss penalty patch*: this kernel is obtained by first applying its suitable PMCTrack patch; then, the scheduler is further modified, as explained in section 3.3 of chapter 3
- *CFS cache-miss reward patch*: this is a negative version of the previous patch, as explained below

In order to be statistically consistent, another kernel version, i.e. *CFS cache-miss reward patch*, has been generated. This kernel is exactly the same as the *CFS cache-miss penalty patch*, with an exception on the underlying logic of penalising or rewarding processes based on high or low cache-miss rate variations relatively to other tasks in the same runqueue. In particular, the reasoning discussed in section 3.1.3 is here just inverted and the relationship in equation 3.3, between cache-miss rate differences and virtual runtime variation, becomes now:

$$\Delta_{VR}^{*} = \left(1 - \frac{\Delta r_p(T)}{\sum_{i=0}^{n-1} |\Delta r_i(T)| + |\Delta r_p(T)|}\right) \cdot \Delta_{VR} \tag{4.1}$$

As opposed to the previous version, higher positive relative variations of a task's cache-miss rate result here in a smaller variation of virtual runtime; this action entails in turn a displacement towards the left of the red-black tree associated with the runqueue, thus increasing the chances for that task to run next. Conversely, for high negative relative variations, the final virtual runtime to be added will be greater, thus moving the process to the right side of the red-black tree. Therefore, this kernel is meant to reward high relative cache-miss rate variations in the scheduling, rather than penalise them as it happened in the previous patch. This inversion in logic is aimed at systematically evaluating the influence of cache-misses rates in the current CFS scheduler.

This reversed behaviour is easily coded by inverting the signs in the last `if-else` statements of function `corr_delta_fair()`, used to compute the variation of `vruntime`:

```c
static u64 corr_delta_fair(u64 delta, s64 del_CMR,
                           u64 run_del_CMR)
{
  ...

  /* Based on the sign of the numerator, add or subtract the percentage */
  if (del_CMR > 0)
    delta = delta-(d1*delta)/10-(d2*delta)/100;
  else
    delta = delta+(d1*delta)/10+(d2*delta)/100;

  return delta;
}
```

The above three kernels are selected for being tested, on top of which a total amount of 600 processes, based on the same programs set, are lauched and let run in single and multi parallel fashion by the test routines *MiBenchSeqLauncher* and *MiBenchParLauncher*, respectively. Furthermore, different sizes for the computational program sections of the generated tasks are also taken into consideration when conducting the experiments, by iterating the specific individual program's instructions inside the wrapper function used to launch each MiBench process. This has been done in order to distinguish among longer and shorter program's code sections, thus differentiating between different average execution times of the considered set of tasks.

At this purpose, 50 and 100 iterations of the programs code sections are evaluated, as well as a standard programs execution, i.e. with just one single iteration. This further differentiation is introduced for the following reason: a physical constraint exists on the sampling period adopted by the monitoring module, which is the time interval between two consecutive readings of the PMCs values. Despite the minimum configurable value is 50 ms, the sampling period has been set to 100 ms in this work, in order to avoid introducing too much overhead in the scheduler [21]. Of course, this choice implies that the proposed altered version of the algorithm is applicable to processes whose execution time is long enough to guarantee that the monitoring module collects at least one sample during each program execution.

From the above considerations, the parameters varying in each experiment performed are the following:

- **Kernel under test:**

  a. *CFS baseline*
  b. *CFS cache-miss penalty patch*
  c. *CFS cache-miss reward patch*

- **Test type:**

  a. Single-parallel execution
  b. Multi-parallel execution

- **Number of iterations of benchmark's code sections:**

  a. $N_i = 1$
  b. $N_i = 50$
  c. $N_i = 100$

- **Input size:**

  a. Small
  b. Large

4.4 Results

Finally, the results of the experiments are here presented. These results show the average execution time of each MiBench process among the set considered, and they are organised in two categories, based on the type of test: single-parallel and multi-parallel execution.

4.4.1 Single-parallel execution

Table 4.1 shows the average execution times, and standard deviations, for the single-parallel execution test running on the baseline (clean) kernel, for both small and large input size (denoted with S and L, respectively) and for three different values of N_i considered.

Program	$N_i = 1$			$N_i = 50$			$N_i = 100$		
	$\mu[ms]$	$\sigma[ms]$	$\sigma[\%]$	$\sigma[ms]$	$\mu[ms]$	$\sigma[\%]$	$\sigma[ms]$	$\mu[ms]$	$\sigma[\%]$
basicmath-S	12.86	4.01	31.2	169.61	1.91	1.1	336.93	3.15	0.9
basicmath-L	55.25	16.40	29.7	1464.88	20.27	1.4	2910.50	32.82	1.1
bitcount-S	46.97	16.26	34.6	1063.89	29.11	2.7	2144.85	46.77	2.2
bitcount-L	293.78	25.21	8.6	12569.66	678.89	5.4	25403.39	1331.08	5.2
qsort-S	19.37	6.51	33.6	245.28	2.52	1.0	463.39	4.12	0.9
qsort-L	87.55	18.88	21.6	3008.43	204.91	6.8	5958.56	454.18	7.6
susan.s-S	36.83	11.33	30.8	700.98	18.23	2.6	1394.35	42.23	3.0
susan.s-L	195.50	31.82	16.3	7865.66	1140.00	14.5	16134.65	2160.48	13.4
susan.e-S	5.21	1.57	30.1	63.63	4.07	6.4	122.26	3.05	2.5
susan.e-L	65.33	21.93	33.6	1635.66	26.40	1.6	3246.73	46.04	1.4
susan.c-S	3.12	0.89	28.5	33.78	1.89	5.6	65.78	2.05	3.1
susan.c-L	36.33	13.20	36.3	705.10	14.43	2.0	1397.11	24.97	1.8

TABLE 4.1: Single-parallel execution on baseline kernel: mean values and standard deviations for different values of N_i.

Figures 4.3, 4.4 and 4.5 show the percentage difference in the final average execution times, obtained by applying the CFS cache-miss penalty and reward patches with respect to the baseline, for the same benchmark set with $N_i = [1, 50, 100]$, respectively. Since each histogram refers to a different value of N_i, the figures highlight the two patches behaviours for three distinct orders of magnitude of the average execution times for the selected programs.

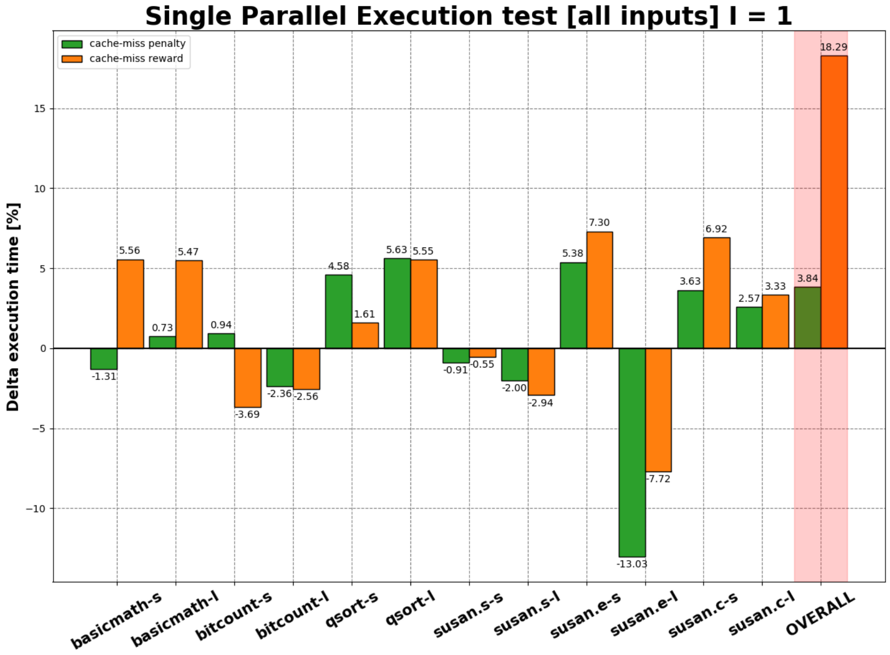


FIGURE 4.3: Single-parallel execution test: percentage difference in execution time for cache-miss penalty (green) and cache-miss reward (orange) patches, with respect to baseline kernel ($N_i = 1$).

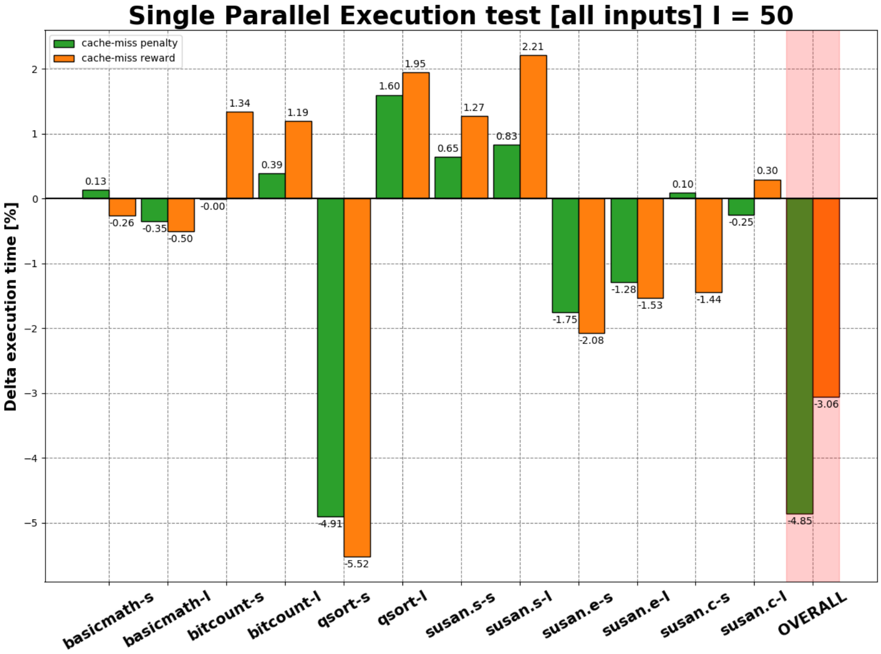


FIGURE 4.4: Single-parallel execution test: percentage difference in execution time for cache-miss penalty (green) and cache-miss reward (orange) patches, with respect to baseline kernel ($N_i = 50$).

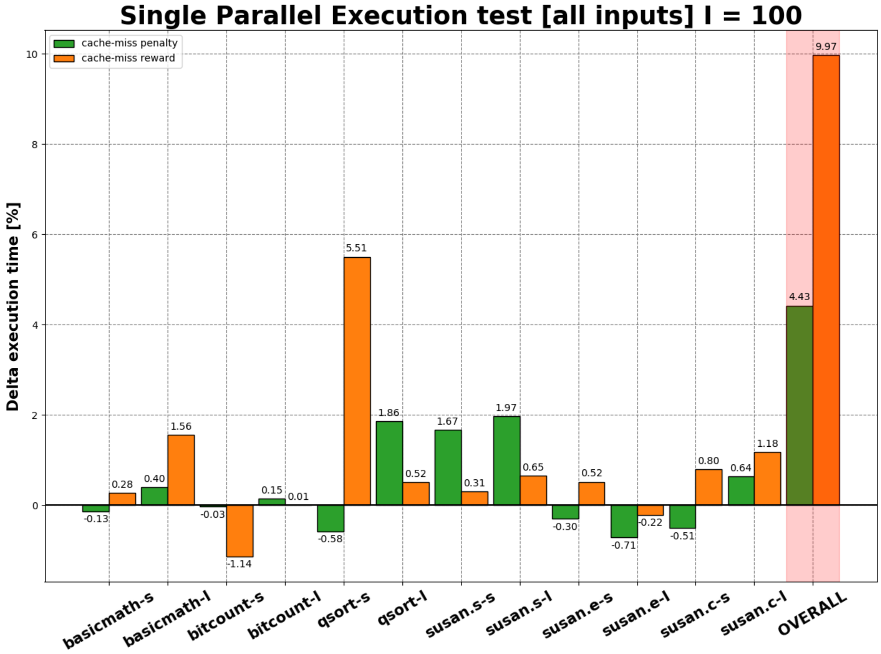


FIGURE 4.5: Single-parallel execution test: percentage difference in execution time for cache-miss penalty (green) and cache-miss reward (orange) patches, with respect to baseline kernel ($N_i = 100$).

For both selected kernel patches, it is now possible to compute a total percentage difference of execution time referred to the whole benchmark, by adding together the values of each bar, taken with their signs. The results are shown in table 4.2.

Kernel patch	$\%\boldsymbol{\Delta T_{exec}}(N_i = 1)$	$\%\boldsymbol{\Delta T_{exec}}(N_i = 50)$	$\%\boldsymbol{\Delta T_{exec}}(N_i = 100)$
cache-miss penalty	+3.85%	-4.84%	+4.43%
cache-miss reward	+18.28%	-3.07%	+9.98%

TABLE 4.2: Single-parallel execution test: total percentage difference of execution time with respect to the baseline kernel, for both cache-miss penalty and reward patches, extended to the whole benchmark.

4.4.2 Multi-parallel execution

Table 4.3 displays the average execution times, and standard deviations, for the multi-parallel execution test running on the baseline (clean) kernel, for both small and large input size and for three different values of N_i considered.

The histograms depicted in 4.6, 4.7 and 4.8 represent the percentage difference in the final average execution times, obtained by applying the CFS cache-miss penalty and reward patches, with respect to the baseline, and considering $N_i = [1, 50, 100]$ for the same set of benchmark processes, respectively. As before, by changing the values of N_i, the graphs describe the two patches behaviours for three distinct orders of magnitude of average execution times for the selected programs.

Benchmark program	$\boldsymbol{N_i = 1}$			$\boldsymbol{N_i = 50}$			$\boldsymbol{N_i = 100}$		
	$\mu[ms]$	$\sigma[ms]$	$\sigma[\%]$	$\mu[ms]$	$\sigma[ms]$	$\sigma[\%]$	$\mu[ms]$	$\sigma[ms]$	$\sigma[\%]$
basicmath-S	3.41	0.18	5.2	168.16	0.49	0.3	336.79	0.92	0.3
basicmath-L	29.62	0.53	1.8	1493.27	3.56	0.2	2985.40	12.17	0.4
bitcount-S	21.87	0.71	3.2	1086.84	5.32	0.5	2170.85	13.78	0.6
bitcount-L	365.92	7.56	2.1	17488.36	147.74	0.8	34926.68	256.09	0.7
qsort-S	5.46	0.18	3.3	250.22	2.06	0.8	511.04	2.72	0.5
qsort-L	64.33	0.74	1.1	3214.22	20.57	0.6	6434.29	36.78	0.6
susan.s-S	13.89	0.43	3.1	703.24	3.87	0.5	1406.68	5.18	0.4
susan.s-L	218.76	2.09	1.0	11173.81	71.79	0.6	22349.55	102.56	0.5
susan.e-S	1.34	0.20	15.0	60.84	0.75	1.2	120.92	0.72	0.6
susan.e-L	33.25	0.56	1.7	1629.49	4.68	0.3	3272.87	7.71	0.2
susan.c-S	0.75	0.04	4.9	32.30	0.42	1.3	64.50	0.59	0.9
susan.c-L	14.22	0.37	2.6	709.13	2.30	0.3	1420.20	3.06	0.2

TABLE 4.3: Multi-parallel execution on baseline kernel: mean values and standard deviations for different values of N_i.

In conclusion, the total percentage difference of execution time referred to the whole benchmark is calculated by adding together the values of each bar, taken with their signs. The results for both cache-miss penalty and reward patches can be found in table 4.4.

Kernel patch	$\%\boldsymbol{\Delta T_{exec}}(N_i = 1)$	$\%\boldsymbol{\Delta T_{exec}}(N_i = 50)$	$\%\boldsymbol{\Delta T_{exec}}(N_i = 100)$
cache-miss penalty	+1.34%	-0.48%	-4.69%
cache-miss reward	+6.49%	+10.86%	+0.66%

TABLE 4.4: Multi-parallel execution test: total percentage difference of execution time with respect to the baseline kernel, for both cache-miss penalty and reward patches, extended to the whole benchmark.

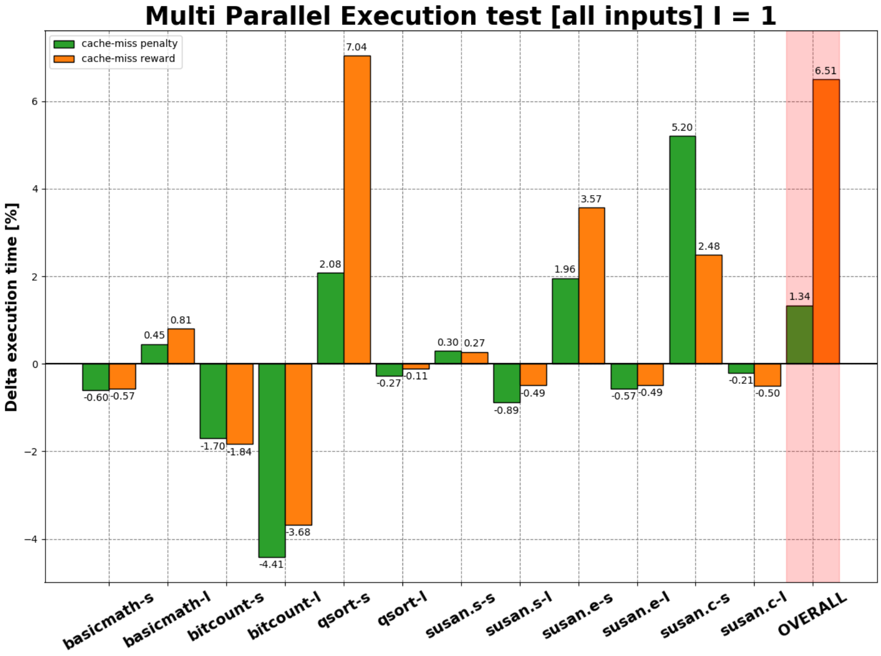


FIGURE 4.6: Multi-parallel execution test: percentage difference in execution time for cache-miss penalty (green) and cache-miss reward (orange) patches, with respect to baseline kernel ($N_i = 1$).

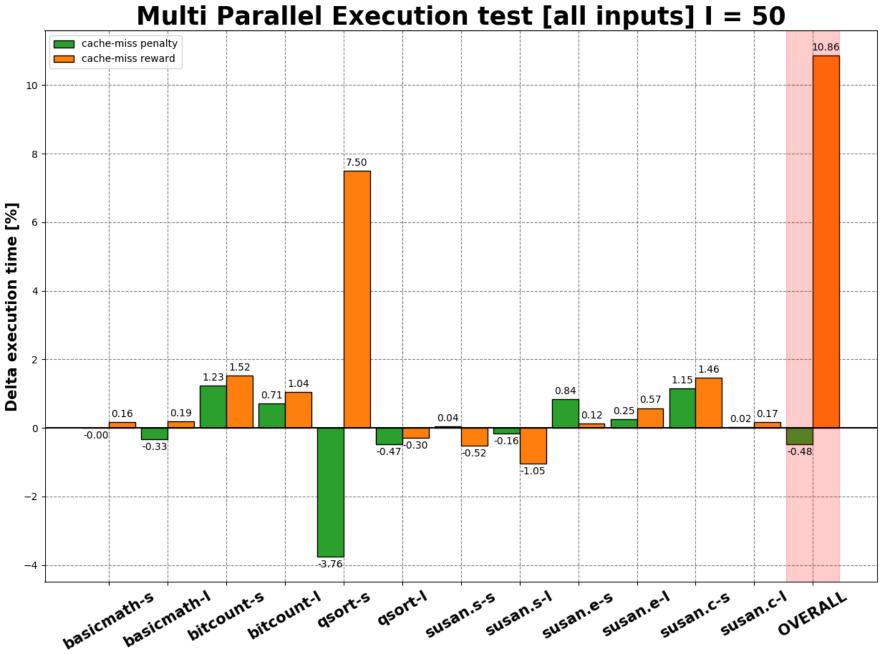


FIGURE 4.7: Multi-parallel execution test: percentage difference in execution time for cache-miss penalty (green) and cache-miss reward (orange) patches, with respect to baseline kernel ($N_i = 50$).

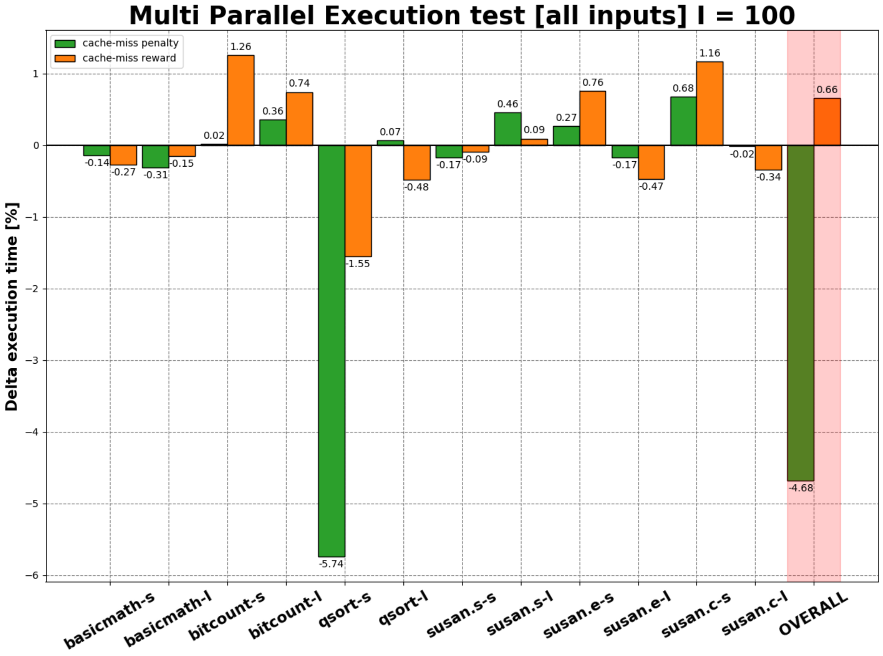


FIGURE 4.8: Multi-parallel execution test: percentage difference in execution time for cache-miss penalty (green) and cache-miss reward (orange) patches, with respect to baseline kernel ($N_i = 100$).

Conclusions

In this thesis, the Linux Completely Fair Scheduler has been properly instrumented to collect dynamic per-task Performance Counters data and use them to alter its algorithm. In particular, this work has been focused on enabling the Linux 4.14.69 kernel to periodically collect per-thread cache-miss rate metrics, obtained by accessing two PMCs programmed to count the number of misses and core cycles, respectively, from whose ratio the desired rate has been computed in units of *misses per cycles*.

After having explored different SW interfaces to access these dedicated HW units, the open source tool PMCTrack has been employed to successfully achieve this goal, by developing a suitable monitoring kernel module [21, 14]. Without loss of generality, this approach has allowed to extend the feature to all tasks in the system, whose scheduling policy is mapped to the CFS algorithm.

The above kernel instrumentation operated by the monitoring module has been used to modify the CFS algorithm and alter its working mechanism. PMCs dynamically collected data have been employed to adjust the value of one of the key variables of the algorithm, i.e. the *virtual runtime*, based on the cache-miss rate of each task. In this way, the cache monitoring information have been used to affect the order by which the runnable processes are selected to run on an available CPU. The sampling period of the monitoring module, i.e. the time interval used by the module to periodically read the counters, has been set to 100 ms, in order to avoid introducing too much overhead in the system [21].

In order to systematically study the influence of the collected metrics on the scheduling, two kernel patches have been implemented: *CFS cache-miss penalty* and *CFS cache-miss reward* patches. The first introduces a penalty or a reward for those tasks whose cache-miss rate, relatively to other tasks in the queue, is growing or reducing, respectively, from one sample collected by the module to another. As appropriate, the patch increases or decreases the virtual runtime of such tasks, thus moving them to the right or the left of the red-black tree runqueue, respectively. Conversely, the second patch acts the other way around with respect to the first, hence it penalises or repays processes whose relative cache-miss rate is decreasing or increasing, respectively.

As testing framework for the above patches, the automotive subset of applications from the MiBench benchmark has been selected [12]. These programs have been wrapped together in a multithreading fashion, in order to develop two test applications, aimed at studying the behaviour of the proposed patch and its counterpart. The first test, i.e. *MiBenchSingleParallelLauncher*, creates a child task for each of the twelve applications belonging to the subset in question; then, it iterates this pattern 50 times, computing a different random sequence of processes per-program at each turn. Instead, the second test *MiBenchMultiParallelLauncher* launches the whole benchmark of applications by generating 50 instances for each program, resulting in 600 processes at a time. Also in this case, the order by which the tasks are created is randomised. In both testing SW, different ranges of execution times for the same set of programs have been also considered, by letting iterate each program section for

$N_i = [1, 50, 100]$ times, in order to simulate medium and long versions of the same benchmark, too.

The results of the single parallel execution test show a slight overall performance boost for the proposed cache-miss penalty patch in the medium-long version of the benchmark, around -5% in total execution time percentage variation. Instead, the same effect appears compensated in the standard and long version of the benchmark, with around +4% of total time percentage difference with respect to the clean kernel. This compensation can be due to a relatively less influence of the proposed policy for standard length of tasks, since the monitoring sampling period is too large with respect to the average execution times in the benchmark; for long average times, instead, it might be due to a stronger cache usage since, if more instructions are processed, the cache gets filled before and instructions and data are more likely to be read from the main memory. However, it is also worth noting that in those two cases, the same test run on the cache-miss reward patch result in a +18% and +10%, respectively, showing a worse trend obtained by rewarding cache-misses, rather then penalising them.

As far as the multi parallel test is concerned, the overall effect on the total execution time difference is not significant for standard and medium lengths of the set of considered programs, while a total boost of -5% is achieved in the long version of the benchmark. For this test, which emphasises the scheduler stress with a larger tasks interleaving, the action of the cache-miss reward patch is generally opposed to the penalty policy, thus suggesting that an increase of cache-miss rates can lead to overall performance degradation.

Several further works can be conducted from this thesis project. First, the proposed patch focuses only on employing cache-miss rates metrics to affect the *order* in which the processes are going to be scheduled; the next step could be to extend the usage of internally collected PMCs information to also modify the *timeslices* assigned to each task by the scheduler.

Another final important consideration is that the main project idea has been employed on the CFS algorithm running on a desktop machine. The analysis could be extended to Linux server systems using the same scheduler, since the performance requirements between the two scenarios are different, i.e. more CPU-bound tasks run on servers, while more interactive applications populate desktop systems.

Furthermore, the monitoring module approach offers great flexibility in terms of portability across different scheduler algorithms, since only one function is used to let the kernel module communicate with the scheduler. In order to completely exploit this strategy, precise measurements on the introduced overhead should be performed, aimed at setting a lower bound for the monitoring sampling period.

Future works may also concern the enlargement of the number and variety of test programs used in the testing framework, based on the requirements and boundaries of future works, while the overall experimental setup has large space for improvements as well. In particular, both tests should be enhanced by enabling them to provide with some insights on the PMCs internal data used in the scheduler, leading to a more controlled and accurate analysis.

Bibliography

[1] Peter Baer Galvin Abraham Silberschatz and Greg Gagne. *Operating Systems Concepts, Ninth Edition*. Danvers, Massachusetts: John Wiley & Sons, Inc., 2012.

[2] Margo Seltzer Alexandra Fedorova and Michael D. Smith. "Cache-Fair Thread Scheduling for Multicore Processors". In: *Harvard Computer Science Group Technical Report TR-17-06* (2006).

[3] Z. F. Baruch. *Structure of computer systems*. Cluj-Napoca: U. T. PRES, 2002.

[4] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. Sebastopol, California: O'Reilly Media, Inc., 2005.

[5] S. Browne et al. "A Portable Programming Interface for Performance Evaluation on Modern Processors". In: *The International Journal of High Performance Computing Applications* 14.3 (2000), pp. 189–204. DOI: 10.1177/109434200001400303. eprint: https://doi.org/10.1177/109434200001400303. URL: https://doi.org/10.1177/109434200001400303.

[6] J. M. Calandrino and J. H. Anderson. "On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler". In: *2009 21st Euromicro Conference on Real-Time Systems*. 2009, pp. 194–204. DOI: 10.1109/ECRTS.2009.13.

[7] William E Cohen. "Tuning programs with OProfile". In: *Wide Open Magazine* 1 (2004), pp. 53–62.

[8] Intel Corporation. *Intel® 64 and IA-32 Architecture's Software Developer's Manual, Vol. 3: System Programming Guide*. Santa Clara, California: Intel Corporation, 2016.

[9] Stijn Eyerman et al. "A Performance Counter Architecture for Computing Accurate CPI Components". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 175–184. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168880. URL: http://doi.acm.org/10.1145/1168857.1168880.

[10] Adrian Garcia-Garcia et al. "LFOC: A Lightweight Fairness-Oriented Cache Clustering Policy for Commodity Multicores". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: ACM, 2019, 14:1–14:10. ISBN: 978-1-4503-6295-5. DOI: 10.1145/3337821.3337925. URL: http://doi.acm.org/10.1145/3337821.3337925.

[11] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. "The Impact of Operating System Scheduling Policies and Synchronization Methods of Performance of Parallel Applications". In: *SIGMETRICS Perform. Eval. Rev.* 19.1 (Apr. 1991), pp. 120–132. ISSN: 0163-5999. DOI: 10.1145/107972.107985. URL: http://doi.acm.org/10.1145/107972.107985.

[12] 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`.

[13] S Jarp, R Jurga, and A Nowak. "Perfmon2: a leap forward in performance monitoring". In: *Journal of Physics: Conference Series* 119.4 (2008), p. 042017. DOI: `10.1088/1742-6596/119/4/042017`. URL: `https://doi.org/10.1088%2F1742-6596%2F119%2F4%2F042017`.

[14] Abel Serrano Roberto Rodríguez-Rodríguez Fernando Castro Daniel Chaver Manuel Prieto-Matias Juan Carlos Saez Jorge Casas. "An OS-Oriented Performance Monitoring Tool for Multicore Systems". In: (Aug. 2015), pp. 697–709.

[15] R. Knauerhase et al. "Using OS Observations to Improve Performance in Multicore Systems". In: *IEEE Micro* 28.3 (2008), pp. 54–66. ISSN: 1937-4143. DOI: `10.1109/MM.2008.48`.

[16] David Koufaty, Dheeraj Reddy, and Scott Hahn. "Bias Scheduling in Heterogeneous Multi-core Architectures". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 125–138. ISBN: 978-1-60558-577-2. DOI: `10.1145/1755913.1755928`. URL: `http://doi.acm.org/10.1145/1755913.1755928`.

[17] T. Li et al. "Efficient operating system scheduling for performance-asymmetric multi-core architectures". In: *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 2007, pp. 1–11. DOI: `10.1145/1362622.1362694`.

[18] Robert Love. *Linux Kernel Development, Third Edition*. RR Donnelley, Crawfordsville, Indiana: Addison-Wesley, 2010.

[19] Wolfang Mauerer. *Professional Linux Kernel Architecture*. Indianapolis, Indiana: Wiley Publishing, Inc., 2008.

[20] Ingo Molnar. *Modular Scheduler Core and Completely Fair Scheduler*. 2007. URL: `https://lwn.net/Articles/230501/`.

[21] 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: `http://oup.prod.sis.lan/comjnl/article-pdf/60/1/60/10329287/bxw065.pdf`. URL: `https://doi.org/10.1093/comjnl/bxw065`.

[22] Karan Singh, Major Bhadauria, and Sally A. McKee. "Real Time Power Estimation and Thread Scheduling via Performance Counters". In: *SIGARCH Comput. Archit. News* 37.2 (July 2009), pp. 46–55. ISSN: 0163-5964. DOI: `10.1145/1577129.1577137`. URL: `http://doi.acm.org/10.1145/1577129.1577137`.

[23] Gautam Upadhyaya Stephen Ziemba and Vijay S. Pai. "Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters". In: (2004).

[24] 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.

[25] *The Unofficial Linux Perf Events Web-Page*. 2011. URL: `http://web.eece.maine.edu/~vweaver/projects/perf_events/index.html`.

[26] *Wiki tutorial on perf*. 2015. URL: `https://perf.wiki.kernel.org/index.php`.