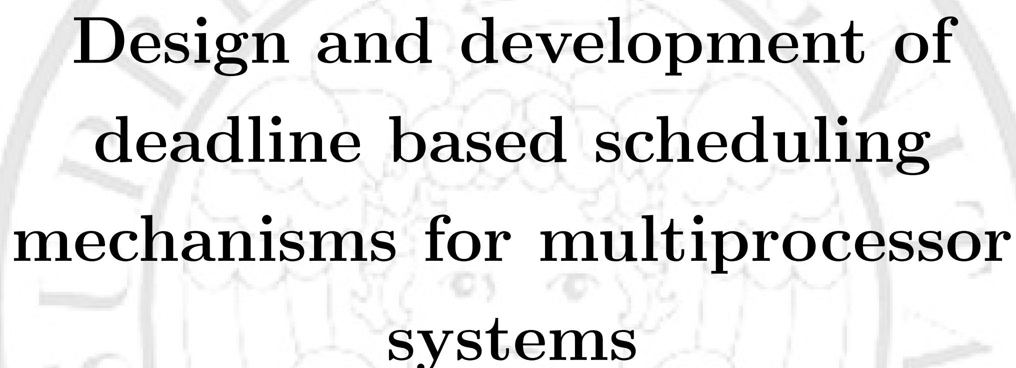
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Thesis and Dissertation Archive - Università di Pisa

Università di Pisa

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea Specialistica



Design and development of deadline based scheduling mechanisms for multiprocessor systems

Relatori:

Prof. Giuseppe Anastasi
Dipartimento di Ingegneria dell'Informazione

Prof. Giuseppe Lipari
Scuola Superiore Sant'Anna

Candidato:

Juri Lelli

Anno Accademico 2009/2010

Alla mia famiglia
e a Claudia.

Abstract

Multiprocessor systems are nowadays de facto standard for both personal computers and server workstations. Benefits of multicore technology will be used in the next few years for embedded devices and cellular phones as well. Linux, as a General Purpose Operating System (GPOS), must support many different hardware platform, from workstations to mobile devices. Unfortunately, Linux has not been designed to be a Real-Time Operating System (RTOS). As a consequence, time-sensitive (e.g. audio/video players) or simply real-time interactive applications, may suffer degradations in their QoS. In this thesis we extend the implementation of the "Earliest Deadline First" algorithm in the Linux kernel from single processor to multicore systems, allowing processes migration among the CPUs. We also discuss the design choices and present the experimental results that show the potential of our work.

Contents

Introduction **vii**

1 Background **1**

1.1 The Linux scheduler 1

1.1.1 Modular scheduling framework 2

1.1.2 Scheduling entities, tasks and runqueues 3

1.1.3 Multiprocessor Data Structures 4

1.2 State of the art of Real-Time scheduling on Linux 6

1.2.1 SCHED_FIFO and SCHED_RR 7

1.2.2 RTLinux, RTAI and Xenomai 8

1.2.3 PREEMPT_RT 9

1.2.4 OCERA 10

1.2.5 AQuoSA 10

1.2.6 FRESCOR 11

1.2.7 LITMUS^{RT} 11

1.3 EDF and CBS theory 12

1.3.1 Earliest Deadline First 12

1.3.2 Constant Bandwidth Server 14

1.4 Caches 15

1.4.1 Organization 16

1.4.2 Functioning 16

1.5 SCHED_DEADLINE implementation 18

1.5.1 Main Features 20

1.5.2 Interaction with Existing Policies 20

1.5.3 Current Multiprocessor Scheduling Support 21
1.5.4 Task Scheduling 22
1.5.5 Usage and Task API 23

2 Design and Development 25
2.1 EDF scheduling on SMP systems 25
2.2 A look at the Linux SMP support 27
2.3 Extending SCHED_DEADLINE 28
2.3.1 Design 28
2.3.2 Development 31

3 Experimental Results 44
3.1 Cases of Interest 44
3.1.1 Working Set Size 44
3.1.2 Cache-Related Overhead 46
3.2 Software Tools 48
3.2.1 Tracing the kernel 48
3.2.2 High resolution clocks 51
3.2.3 Performance Counters 52
3.2.4 Tasks Sets Generator 57
3.3 Hardware 59
3.4 Results 60
3.4.1 Working Set Size Selection 60
3.4.2 Cache-Related Overhead 64
3.4.3 Experiments on Random Tasks Sets 65

4 Conclusions and Future Works 73

A Source Code 76
A.1 SCHED_DEADLINE 76
A.2 Micro-benchmarks 76

Bibliography 83

List of Figures

1.1 The Linux modular scheduling framework. 3
1.2 The CFS runqueue. 5
1.3 An EDF schedule example. 13
1.4 Example of a multi-core CPU cache hierarchy. 17
1.5 The Linux modular scheduling framework with `SCHED_DEADLINE`. 21

3.1 An example of a random tasks set. 58
3.2 Single thread sequential access with elements of 64 bytes. . . . 61
3.3 Single thread sequential access with elements of 32 bytes. . . . 62
3.4 Single thread *random walk* with elements of 64 bytes. 63
3.5 Worst case per-item access time estimate (16KB). 64
3.6 Worst case per-item access time estimate (400KB). 65
3.7 Slack time / Period for G-EDF and P-EDF. 68
3.8 L1D cache misses / execution time for G-EDF and P-EDF. . . 69
3.9 Percentage error between real and expected execution time for G-EDF and P-EDF. 70
3.10 Slack time / Period for G-EDF and G-RM. 70
3.11 L1D cache misses / Execution time for G-EDF and G-RM. . . 71
3.12 Percentage error between Real and Demanded Execution time for G-EDF and G-RM. 72

Introduction

Multiprocessor systems are nowadays de facto standard for both personal computers and server workstations. Benefits of dual-core technology will be used in the next few years for embedded devices and cellular phones as well. Increases in computational power is not the answer for overall better performance, as recently stated by Rob Coombs, director of mobile solution for ARM, in an interview [40] : "We don't need silly GHz speeds. With our dual core A9, we can get two times the performance, without the speed draining the battery so by the time you get home your phone is dead.".

Linux, as a General Purpose Operating System (GPOS), should be able to run on every possible system, from workstations to mobile devices. Even if each configuration has is own issues, the at large trend seems to be a considerable interest in using Linux for real-time and control applications.
Linux has not been designed to be a Real-Time Operating System (RTOS) and this imply that a classical real-time feasibility study of the system under development is not possible, there's no way to be sure that timing requirements of tasks will be met under *every* circumstance. POSIX-compliant fixed-priority policies offered by Linux are not enough for specific application requirements.
Great issues arise when size, processing power, energy consumption and costs are tightly constrained. Time-sensitive applications (e.g., MPEG players) for embedded devices have to efficiently make use of system resources and, at the same time, meet the real-time requirements.
Modified versions of the Linux kernel[1] with improved real-time support are marketed [25, 41, 30], but these are often non-free and cannot take advantage

[1] Operating System's core.

of the huge development community of the standard kernel.

In a recent paper [20] Dario Faggioli and others proposed an implementation of the "Earliest Deadline[2] First" (EDF) [28, 39] algorithm in the Linux kernel. With the words of the authors: "...we believe that to be really *general*, Linux should also provide enhanced real-time scheduling[3] capabilities". EDF is a well known real-time dynamic-priority scheduling algorithm based on a simple concept: the process with the earliest deadline will be the first to run. In order to extend stock Linux kernel's features a new scheduling policy has been created: `SCHED_DEADLINE`. At the time of writing a complete patch set against vanilla kernel[4] is under evaluation by the Linux development community [19]. `SCHED_DEADLINE` works natively on multicore platform but is at present completely partitioned: each CPU has is own runqueue and is treated as an essentially independent scheduling domain[5]. Lacking the possibility of migrate[6] processes between CPUs, the assignment of tasks to CPUs can be a pain and it is hard (or impossible) to get full utilization of the system.
In addition to what previously stated it has to be said that some simple scheduling problems cannot be solved without process migration. We can, for example, imagine a two-CPU system running three processes, each of which needs 60% of a single CPU's time. The system has the resources to run those three processes, but not if it is unable to move processes between CPUs [11].

In this thesis we extend `SCHED_DEADLINE` scheduling policy to allow processes migrations. The at first sight solution is indeed a global runqueue at which CPUs can draw on in order to decide which process to run. This implementation is practically unfeasible due to the scalability problems that entails. So we have decided to follow what `sched_rt` group has done [35]. Our choice has been to leave runqueues on CPUs and actively push and pull

[2]The instant of time at which the work must be completed.

[3]The *scheduler* is a component of the kernel that selects which process to execute at any instant of time and is responsible of dividing the finite resource of processor time between all runnable processes in the system.

[4]Stock Linux kernel maintained by Linus Torvalds.

[5]A `sched_domain` contains critical informations for the scheduler.

[6]Move a process from a CPU to an another.

processes from these as needed. The goal is to have at any instant of time the m earliest deadline processes running on the m CPUs of the system. Moreover we present experimental results that try to prove the goodness of our work both in terms of performance and scheduling/hardware overhead.

This document is organized as follows.

Chapter 1 (**Background**) gives a brief overview of the concepts and the theory on which this thesis is based. First, the modular framework of the Linux scheduler is analyzed (with special attention to multiprocessors systems), then we find the state of the art of Real-Time scheduling on Linux. Since we will extend the EDF/CBS implementation contained inside `SCHED_DEADLINE` scheduling policy, in this chapter we also give some insights on the theory behind those real-time scheduling algorithms and analyze how they are implemented inside the Linux kernel. Cache memories organization and functioning are depicted as well in order to be able to present some of the outcomes of the conducted experiments.

Chapter 2 (**Design an Development**) is the "main course" of this work. The theoretical aspects of EDF scheduling on multiprocessors are presented in the first section of the chapter, leaving to the second one a look at how Linux behaves in SMP systems. We then present design choices we made and, in the last section, we will dive deep into the code in order to explain how the most interesting parts of the push/pull logic implementation works.

In Chapter 3 (**Experimental Results**) we present the results of the experiments we conducted in order to analyse the functioning of our work. This chapter also contains a section in which we explain how we collected measures and how the programs built for experimental purposes works.

Finally, in Chapter 4 (**Conclusions**), we sum up results and suggest possible future extensions to the code and alternate ways of testing.

Chapter 1

Background

1.1 The Linux scheduler

The process scheduler is the component of the kernel that selects which process to run next. Processor time is a finite resource, the process scheduler (or simply the *scheduler*) is a subsystem of the kernel that divides processor time between the runnable processes. Working in a single processor machine the scheduler has to give the impression to the user that multiple processes are executing simultaneously. This is the basis of a *multitasking*[1] operating system like Linux.
On multiprocessor machines processes can actually run concurrently (in parallel) on different processors. The scheduler has to assign runnable processes to processors and decide, on each of them, which process to run.

How the scheduler works affect how the system behaves. We can privilege task switching in order to have a reactive and interactive system, we can allow tasks to run longer and have a batch jobs well suited system, we can also decide that some tasks are vital for the system and must execute to the detriment of the others.

[1] In this context *task* and *process* are used as synonyms.

1.1.1 Modular scheduling framework

The Linux scheduler has been designed and implemented by Ingo Molnar [29] as a modular framework that can easily be extended. Each scheduler module is a *scheduling class* that encapsulate specific scheduling poilicies details about which the core scheduler is in the dark.
Scheduling classes are implemented through the `sched_class`[2] structure, which contains hooks to functions that must be called whenever the respective event occurs. A (partial) list of the hooks follows:

- `enqueue_task(...)`: called when a task enters a runnable state. It enqueues a task in the data structure used to keep all runnable tasks (runqueue, see below).
- `dequeue_task(...)`: called when a task is no longer runnable. It removes a task from the runqueue.
- `yield_task(...)`: yields the processor giving room to the other tasks to be run.
- `check_preempt_curr(...)`: checks if a task that entered the runnable state should preempt the currently running task.
- `pick_next_task(...)`: chooses the most appropriate task eligible to be run next.
- `put_prev_task(...)`: makes a running task no longer running.
- `select_task_rq(...)`: chooses on which runqueue (CPU) a waking-up task has to be enqueued.
- `task_tick(...)`: mostly called from the time tick functions, it executes running task related periodical stuff.

In the Linux scheduler are at present implemented three *"fair"* (`SCHED_NORMAL`, `SCHED_BATCH`, `SCHED_IDLE`) and two *real-time* (`SCHED_RR`, `SCHED_FIFO`) scheduling policies. This situation can be better view with Figure 1.1 on the following page.

[2] Defined in `include/linux/sched.h`.

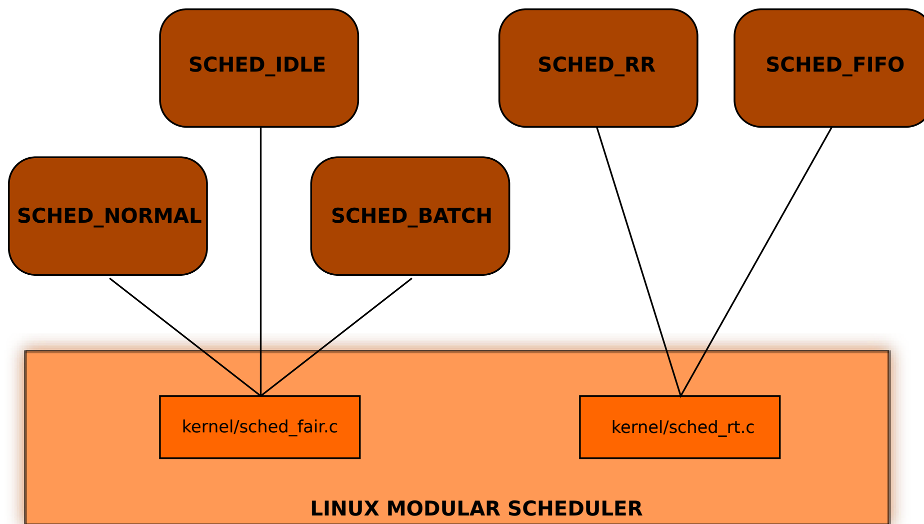


Figure 1.1: The Linux modular scheduling framework.

1.1.2 Scheduling entities, tasks and runqueues

All the things that the scheduler uses to implement scheduling policies are contained into `struct sched_entity`[3] (there is a *scheduling entity* for each scheduler module). Looking inside that structure we find the fields (e.g. `exec_start`, `vruntime`, etc...) that the CFS[4] scheduler uses to do his job.
The concept of *scheduling entity* is essentially "something to be scheduled", which might not be a process (there also exist tasks groups [10]).

At the very beginning of `struct task_struct`[5] there are the fields that distinguish tasks. Among others:

- `volatile long state`: describes task's state, it can assume three values (`-1`, `0`, `>0`) depending on the task respectively beeing *unrunnable*, *runnable* or *stopped*.
- `const struct sched_class *sched_class`: binds the task with his scheduling class.

[3]Defined in `include/linux/sched.h`.
[4] *Completely Fair Scheduler*, the default Linux scheduler, see [15].
[5]Defined in `include/linux/sched.h`.

- `struct sched_entity se`, `struct sched_rt_entity rt`: contain *scheduling entity* related informations.
- `cpumask_t cpus_allowed`: mask of the cpus on which the task can run.
- `pid_t pid`: process identifier that uniquely identifies the task.

Last but not least come runqueues. Linux has a main per-CPU runqueue data structure called not surprisingly `struct rq`[6]. Runqueues are implemented in a modular fashion as well. The main data structure contains a "sub-runqueue" field for each scheduling class, and every scheduling class can implement his runqueue in a different way.
It is enlightening to look at the CFS runqueue implementation. Structure `struct cfs_rq` holds both accounting informations about enqueued tasks and the actual runqueue. CFS uses a time-ordered red-black tree to enqueue tasks and to build a "timeline" of future task execution.
A red-black tree is a type of self-balancing binary search tree. For every running process there is a node in the red-black tree. The process at the leftmost position is the one to be scheduled next. The red-black tree is complex, but it has a good worst-case running time for its operations and is efficient in pratice: it can search, insert and delete in $O(\log n)$ time, where n is the number of elements in the tree. The leaf nodes are not relevant and do not contain data. To save memory, sometimes a single sentinel node performs the role of all leaf nodes.
Scheduling class designers must cleverly choose a runqueue implementation that best fits scheduling policies needs. Figure 1.2 on the next page shows all this at first sight confusing things.

1.1.3 Multiprocessor Data Structures

Since now we haven't talked about how many processor our system has. In fact all that we have said remains the same for uni-processor and multi-processor machines as well.

[6]Defined in `kernel/sched.c`, as other runqueue related things.

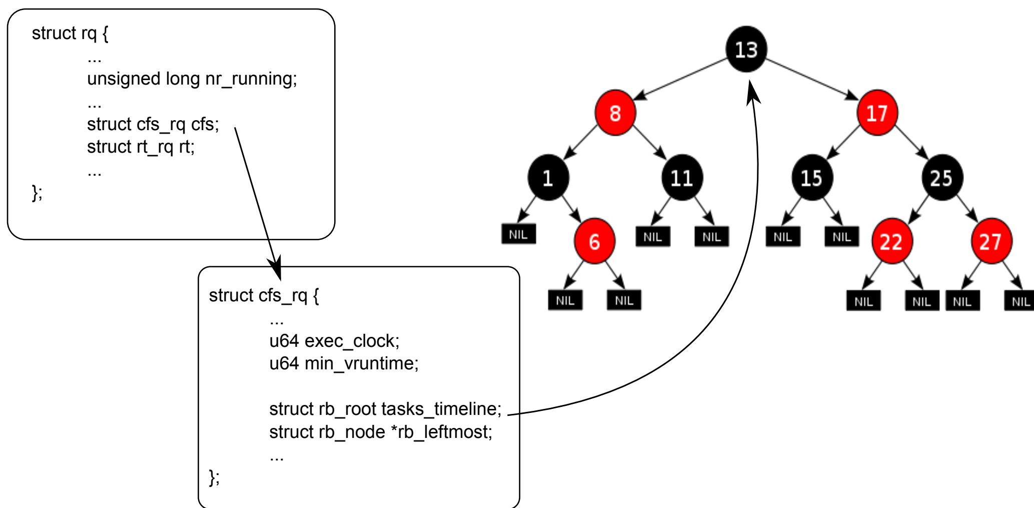


Figure 1.2: The CFS runqueue.

A multiprocessor Linux kernel (one configured with `CONFIG_SMP`) has additional fields into the afore-mentioned structures in comparison to a uniprocessor one. In `struct sched_class` we find:

- `select_task_rq(...)`: called from `fork`, `exec` and wake-up routines; when a new task enters the system or a task is waking up the scheduler has to decide which runqueue (CPU) is best suited for it.
- `load_balance(...)`: checks the given CPU to ensure that it is balanced within scheduling domain (see below); attempts to move tasks if there is an imbalance. It is important to say that this function is not implemented by every scheduling class, the reason why will be explained later.
- `pre_schedule(...)`: called inside the main `schedule` routine; performs the scheduling class related jobs to be done before the actual schedule.
- `post_schedule(...)`: like the previous routine, but after the actual schedule.
- `task_woken(...)`: called when a task wakes up, there could be things to do if we are not going to schedule soon.

- `set_cpus_allowed(...)`: changes a given task's CPU affinity; depending on the scheduling class it could be responsible for to begin tasks migration.

A modern large multiprocessor system can have a complex structure and, at-large, processors have unequal relationships with each other. Virtual CPUs of a hyperthreaded core share equal access to memory, cache and even the processor itself. On a symmetric multiprocessing system (SMP) each processor maintains a private cache, but main memory is shared. Nodes of a NUMA architecture have different access speeds to different areas of main memory. To get things worse all these options can coexist: each NUMA node looks like an SMP system which may be made up of multiple hyperthreaded processors. One of the key jobs a scheduler must do on a multiprocessor (non real-time) system is balancing the load across the CPUs. Teaching the scheduler to migrate tasks intelligently under many different types of loads is not so easy. In order to cope with this problem *scheduling domains* [12] have been introduced into the Linux kernel.
A *scheduling domain* (`struct sched_domain`[7]) is a set of CPUs which share properties and scheduling policies, and which can be balanced against each other. Scheduling domains are hierarchical, a multi-level system will have multiple levels of domains. A struct pointer `struct sched_domain *sd`, added inside `struct rq`, creates the binding between a runqueue (CPU) and his scheduling domain. Using scheduling domain informations the scheduler can do a lot to make good scheduling and balancing decisions.

1.2 State of the art of Real-Time scheduling on Linux

Linux has been designed to be a general-purpose operating system (GPOS), therefore it presents some issues, like unpredictable latencies, limited support for real-time scheduling, and coarse-grain timing resolution that might be a problem for real-time application [26]. The main design goal of the Linux

[7]Defined in `include/linux/sched.h`.

kernel has been (and still remains) to optimize the average throughput (i.e., the amount of "useful work" done by the system in the unit of time).
During the last years, research institutions and independent developers have proposed several real-time extensions to the Linux kernel. In this section we present a brief description of the more interesting alternatives.

1.2.1 SCHED_FIFO and SCHED_RR

Since Linux is a POSIX-compliant operating system, the Linux scheduler must provide `SCHED_FIFO` and `SCHED_RR`, simple fixed-priority policies. As the standard states[8], `SCHED_FIFO` is a strict first in-first out (FIFO) scheduling policy. This policy contains a range of at least 32 priorities (actually 100 inside Linux). Threads (tasks) scheduled under this policy are chosen from a thread list ordered according to the time its threads have been in the list without being executed. The head of the list is the thread that has been in the list the longest time; the tail is the thread that has been in the list the shortest time.
`SCHED_RR` is a round-robin scheduling policy with a per-system time slice (time quantum). This policy contains a range of at least 32 priorities and is identical to the `SCHED_FIFO` policy with an additional condition: when the implementation detects that a running process has been executing as a running thread for the time quantum, or longer, the thread becomes the tail of its thread list, and the head of that thread list is removed and made a running thread.
Both `SCHED_FIFO` and `SCHED_RR` unfortunately diverges from what the real-time research community refer to as "realtime" [7]. As best-effort policies, they do not allow to assign timing constraints to tasks. Other notable drawbacks of fixed priority schedulers are the fairness and the security among processes [3]. In fact, if a regular non-privileged user is enabled to access the real-time scheduling facilities, then he can also rise his processes to the highest priority, starving the rest of the system.

[8]IEEE Std 1003.1b-1993

1.2.2 RTLinux, RTAI and Xenomai

RTLinux is a patch developed at *Finite State Machine Labs* (FSMLabs) to add realt-time features to the standard Linux kernel [43]. The RTLinux patch implements a small and fast RTOS, utilizing the *Interrupt Abstraction* approach. The approach based on Interrupt Abstraction consists of creating a layer of virtual hardware between the standard Linux kernel and the computer hardware (*Real-Time Hardware Abstraction Layer*). The RTHAL actually virtualizes only interrupts. To give an idead of how it works (a complete description is beyond the focus of this thesis) we can imagine that the RT-kernel and the Linux kernel work side by side. Every interrupt source coming from real hardware is marked as real-time or non real-time. Real-time interrupts are served by the real-time subsystem, whereas non-real-time interrupts are managed by the Linux kernel. In pratice, the resulting system is a multithreaded RTOS, in which the standard Linux kernel is the lowest priority task and only executes when there are no real-time tasks to run and the real-time kernel is inactive.

RTAI is the acronym of "*Real-Time Application Interface*" [36]. The project started as a variant of RTLinux in 1997 at Dipartimento di Ingegneria Areospaziale of Politecnico di Milano (DIAPM), Italy. Although the RTAI project started from the original RTLinux code, the API of the projects evolved in opposite directions. In fact, the main developer (prof. Paolo Mantegazza) has rewritten the code adding new features and creating a more complete and robust system. The RTAI community has also developed the *Adaptive Domain Environment for Operating Systems* (ADEOS) nanokernel as alternative for RTAI's core to exploit a more structured and flexible way to add a real-time environment to Linux [16]. The ADEOS nanokernel implements a pipeline scheme into which every domain (OS) has an entry with a predefined priority. RTAI is is the highest priority domain which always processes interrupts before the Linux domain, thus serving any hard real time activity either before or fully preempting anything that is not hard real time.

Xenomai [22] is a spin-off of the RTAI project that brings the concept of virtualization one step further. Like RTAI, it uses the ADEOS nanokernel to

provide the interrupt virtualization, but it allows a real-time task to execute in user space extensively using the concept of domain provided by ADEOS (also refer to [26] for a deeper insight).

All the alternatives before are efficient solutions, as they allow to obtain very low latencies, but are also quite invasive, and, often, not all standard Linux facilities are available to tasks running with real-time privileges (e.d., Linux device drivers, network protocol stacks, etc. . .). Another major problem (on RTLinux and RTAI) is that the real-time subsystem executes in the same memory space and with the same privileges as the Linux kernel code. This means that there is no protection of memory between real-time tasks and the Linux kernel; a real-time task with errors may therefore crash the entire system.

1.2.3 PREEMPT_RT

The `CONFIG_PREEMPT_RT` [24] patch set is maintained by a small group of core developers, led by Ingo Molnar. This patch allows nearly all of the kernel to be preempted, with the exception of a few very small regions of code. This is done by replacing most kernel spinlocks with mutexes that support *priority inheritance*, as well as moving all interrupt and software interrupts to kernel threads.
The Priority Inheritance (PI) protocol solves the problem of unbounded *priority inversion*. You have a priority inversion when a high priority task must wait for a low priority task to complete a critical section of code and release the lock. If the low prioriry task is preempted by a medium priority task while holding the lock, the high priority task will have to wait for a long (unbounded) time. The *priority inheritance protocol* dictates that in this case, the low priority task *inherits* the priority of the high priority task while holding the lock, preventing the preemption by medium priority tasks.
The `CONFIG_PREEMPT_RT` patch set focus is, in short, make the Linux kernel more deterministic, by improving some parts that do not allow a predictable behaviour. Even if the priority inheritance mechanism is a complex algorithm to implement, it can help reduce the latency of Linux activities, reaching the

level of the *Interrupt Abstraction* methods [26].

1.2.4 OCERA

OCERA [33], that stands for Open Components for Embedded Real-time Applications, is an European project, based on Open Source, which provides an integrated execution environment for embedded real-time applications. It is based on components and incorporates the latest tecniques for build embedded systems.
A real-time scheduler for Linux 2.4 has been developed within this project, and it is availabe as open source code [3], [34], [37]. To minimize the modifications to the kernel code, the real-time scheduler has been developed as a small patch and an external loadable kernel module. All the patch does is exporting toward the module (by some *hooks*) the relevant scheduling events. The approach is straightforward and flexible, but the position where the hooks have to be placed is real challenge, and it made porting the code to next releases of the kernel very hard.

1.2.5 AQuoSA

The outcome of the OCERA project gave birth to the AQuoSA [4] software architecture. AQuoSA is an open-source project for the provisioning of *adaptive Quality of Service* functionality into the Linux kernel, developed at the Real Time Systems Laboratory of Scuola Superiore Sant'Anna. The project features a flexible, portable, lightweight and open architecture for supporting soft real-time applications with facilities related to timing guarantees and QoS, on the top of a general-purpose operating system as Linux.
It basically consists on porting of OCERA kernel approach to 2.6 kernel, with a user-level library for feedback based scheduling added. Unfortunately, it lacks features like support for multicore platforms and integration with the latest modular scheduler (see section 1.1.1).

1.2.6 FRESCOR

FRESCOR [21] is a consortium research project funded in part by the European Union's Sixth Framework Programme [18]. The main objective of the project is to develop the enabling technology and infrastructure required to effectively use the most advanced techniques developed for real-time applications with flexible scheduling requirements, in embedded systems design methodologies and tools, providing the necessary elements to target reconfigurable processing modules and reconfigurable distributed architectures.
A real-time framework based on Linux 2.6 has been proposed by this project. It is based on AQuoSA and further adds to it a contract-based API and a complex middleware for specifying and managing the system performances, from the perspective of the Quality of Service it provides. Obviously, it suffers from all the above mentioned drawbacks as well.

1.2.7 LITMUS^{RT}

The LITMUS^{RT} [27] project is a soft real-time extension of the Linux kernel with focus on multiprocessor real-time scheduling and synchronization. The Linux kernel is modified to support the sporadic task model and modular scheduler plugins. Both partitioned and global scheduling is supported.
The primary purpose of the LITMUS^{RT} project is to provide a useful experimental platform for applied real-time systems research. In that regard LITMUS^{RT} provides abstractions and interfaces within the kernel that simplify the prototyping of multiprocessor real-time scheduling and synchronization algorithms.
LITMUS^{RT} is not a production-quality system, is not "stable", POSIX-compliance is not a goal and is not targeted at being merged into mainline Linux. Moreover, it only runs on Intel (x86-32) and Sparc64 architectures (i.e., no embedded platforms, the one typically used for industrial real-time and control).

1.3 EDF and CBS theory

In order to understand the choices made on implementing `SCHED_DEADLINE` scheduling class, we present here a brief discussion of the theory behind those. For this purpose will be used the following notation:

τ_i identifies a generic periodic task;

ϕ_i identifies the *phase* of task τ_i; i.e., the first instance activation time;

T_i identifies the *period* of task τ_i; i.e., the interval between two subsequent activations of τ_i;

C_i identifies the *Worst-Case Execution Time* (WCET) of task τ_i;

D_i identifies the relative deadline of task τ_i; a symplifying assumption is that $D_i = T_i$;

$d_{i,j}$ identifies the absolute deadline of the j-th job of task τ_i; it can be calculated as $d_{i,j} = \phi_i + (j-1)T_i + D_i$;

U identifies the CPU utilization factor; it is calculated as $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$, and provides a measure of CPU load by a set of periodic tasks.

1.3.1 Earliest Deadline First

Dynamic priority algorithms are an important class of scheduling algorithms. In these algorithms the priority of a task can change during its execution. In fixed priority algorithms (a sub-class of the previous one), instead, the priority of a task does not change throughout its execution.
Earliest Deadline First (EDF) schedules tasks for increasing absolute deadline. At every instant of time, the selected task from the runqueue is the one with the earliest absolute deadline. Since the absolute deadline of a periodic task depends from the k-th current job,

$$d_{i,j} = \phi_i + (j-1)T_i + D_i,$$

EDF is a dynamic priority algorithm. In fact, although the priority of each job is fixed, the relative priority of one task compared to the others varies over time.
EDF is commonly used with a preemptive scheduler, when a task with an earlier deadline than that of the running task gets ready the latter is suspended and the CPU is assigned to the just arrived earliest deadline task. This algorithm can be used to schedule periodic and aperiodic tasks as well, as task selection is based on absolute deadline only.
A simple example may clarify how EDF works (figure 1.3). A task set composed by three tasks is scheduled with EDF: $\tau_1 = (1,4)$, $\tau_2 = (2,6)$, $\tau_3 = (3,8)$, with $\tau_i = (C_i, T_i)$. The utilization factor is: $U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = \frac{23}{24}$. All three tasks arrive at instant 0. Task τ_1 starts execution since it has the earliest deadline. At instant 1, τ_1 has finished his job and τ_2 starts execution; the same thing happens at instant 3 between τ_2 and τ_3. At instant 4, τ_1 is ready again, but it does not start executing until instant 6, when becomes the earliest deadline task (*ties can be broken arbitrarly*). The schedule goes on this way until instant 24 (*hyperperiod*, least common multiple of tasks periods), then repeats the same.

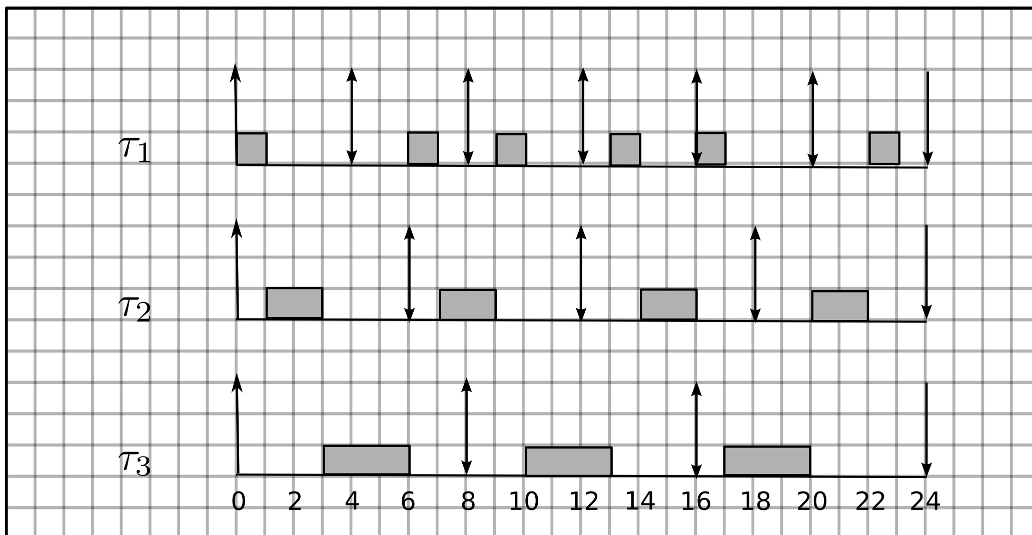


Figure 1.3: An EDF schedule example.

Last thing to say is about schedulability bound with EDF:

- **Theorem** [28]: given a task set of periodic or sporadic tasks, with relative deadlines equal to periods, the task set is schedulable by EDF if and only if
$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1.$$
- **Corollary**: EDF is an *optimal algorithm* on preemptive uniprocessor systems, in the sense that if a task set is schedulable, it is schedulable by EDF (you can reach a CPU utilization factor of 100%).

We could ensure the schedulability of the task set in fig. 1.3 simply considering that $U = \frac{23}{24} \leq 1$.

1.3.2 Constant Bandwidth Server

In section 1.3.1 we have considered homogeneous task set only (periodic or aperiodic). Here we have to cope with scheduling a task set composed by periodic and aperiodic tasks as well. Periodic tasks are generally considered of a hard type, whereas aperiodic tasks may be hard, soft or even non real-time, depending on the application.
Using a periodic task (*server*), dedicated to serve aperiodic requests, is possible to have a good average response time of aperiodic tasks. As every periodic task, a server is characterized by a period T_s and a computing time C_s, called server *budget*. A server task is scheduled with the same algorithm used for periodic tasks, and, when activated, serves the hanging aperiodic requests (not going beyond its C_s).
The Constant Bandwidth Server (CBS) [2, 1] is a service mechanism of aperiodic requests on a dynamic context (periodic tasks are scheduled with EDF) and can be defined as follows:

- A CBS is characterized by an ordered pair (Q_s, T_s) where Q_s is the maximum budget and T_s is the period of the server. The ratio $U_s = Q_s/T_s$ is denoted as the server bandwidth.

- The server manages two internal variables that define its state: c_s is the current budget a time t (zero-initialized) and d_s is the current deadline assigned by the server to a request (zero-initialized).
- If a new request arrives while the current request is still active, the former is queued in a server queue (managed with an arbitrary discipline, for example FIFO).
- If a new request arrives at instant t, when the server is idle, you see if you can recycle current budget and deadline of the server. If it is $c_s \leq (t - d_s)U_s$, then we can schedule the request with the current server values, else we have to replenish the budget with the maximum value ($c_s = Q_s$) and calculate the deadline as $d_s = t + T_s$.
- When a request is completed, the server takes the next (if it exists) from the internal queue and schedule it with the current budget and deadline.
- When the budget is exhausted ($c_s = 0$), it is recharged at the maximum value ($c_s = Q_s$) and the current deadline is postponed of a period ($d_s = d_s + T_s$).

The basic idead behind the CBS algorithm is that when a new request arrives it has a deadline assigned, which is calculated using the server bandwidth, and then inserted in the EDF ready queue. In the moment an aperiodic task tries to execute more than the assigned server bandwidth, its deadline gets postponed, so that its EDF priority is lowered and other tasks can preempt it.

1.4 Caches

One of the great concern of the real-time research community, at the time of writing, is how a global scheduler behaves considering modern processor's *cache memories*. We have seen in the precedent section that a careful estimate of WCET is fundamental to guarantee a task set schedulability. In

SMP systems the WCET of a task may be unexpectedly inflated by the fact that system's caches are a shared resource, all the running task potentially competing for it. We give here a brief overview of cache organization and functioning in order to understand the experimental case studies.

1.4.1 Organization

Modern processors employ a hierarchy of fast *cache memories* that contain recently-accessed instructions and operands to alleviate high off-chip memory latencies (as the main memory latency). Caches are organized in levels; proceding from the top to the bottom, we usually find a per-processor *private* Level-1 (L1) cache (the fastest and smallest among caches), then a variable number of deeper caches (L2, L3, etc. . .) being successively larger and slower. Depending on the particular CPU, cache levels (except L1) may be *shared* among all the cores or used only by exclusive subsets of them (figure 1.4).

A cache contains either instructions or data (as in the L1 case, where we find Level-1 Data (L1D) and Level-1 Instructions (L1I) caches), and may contain both if it is *unified*.

Caches operate on blocks of consecutive addresses called *cache lines* with common sizes ranging from 8 to 128 bytes. The loading of data from a lower level cache (or from main memory) always proceeds at cache line size chunks, even if a smaller amount of data is needed. The *associativity* of a cache determines how the full main memory address space is mapped to the small cache address space. In *direct mapped* caches, each location in main memory can be cached by just one cache location. In *fully associative* caches, on the contrary, each location in main memory may reside at any location in the cache. On modern systems, in fact, most caches are *set associative*, wherein each memory location may reside at a fixed number of places.

1.4.2 Functioning

The data set accessed by a job instance in doing its work is traditionally called the *working set size* (WSS) of the job. Cache lines are loaded into the cache when needed only. When a job references a cache line that cannot be

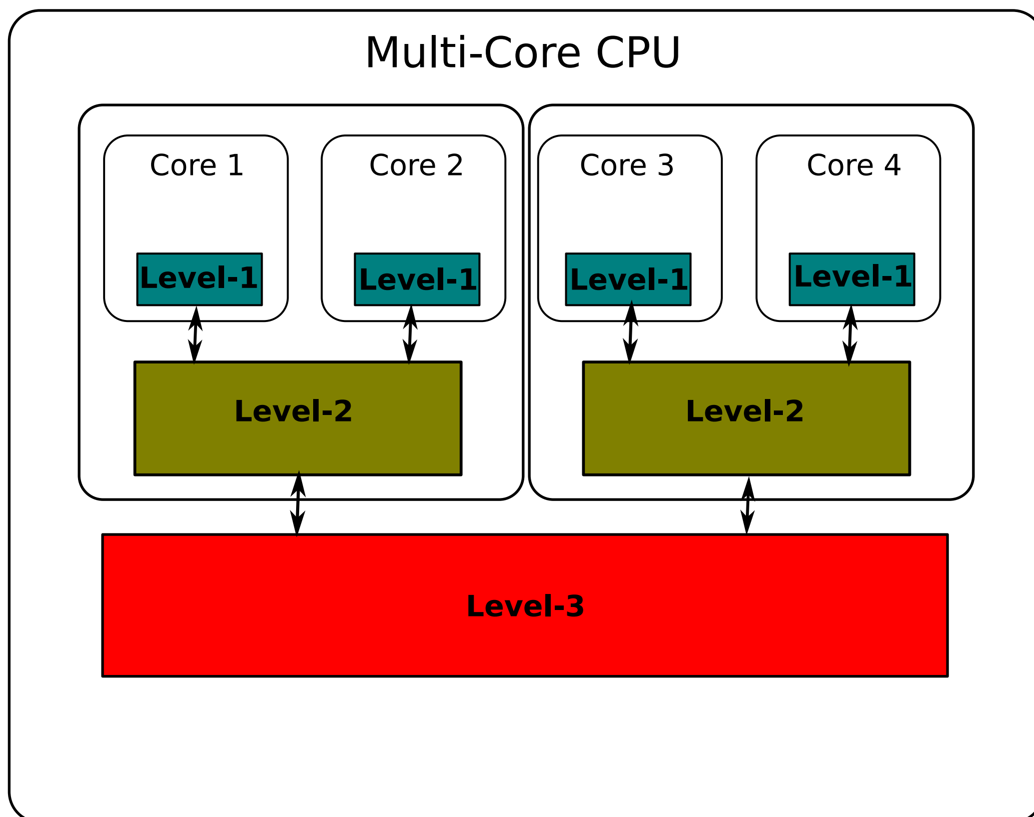


Figure 1.4: Example of a multi-core CPU cache hierarchy.

found in a level-X cache, then it suffers a *level-X cache miss*. Cache misses are also distinguished by the reason that caused them:

- *Compulsory misses*: are triggered the first time a cache line is referenced. Before a job can work on data, they must be present on some cache location.

- *Capacity misses*: result if the WSS of the job exceeds the size of the cache.

- *Conflict misses*: arise if useful cache lines were evicted to make room of othere cache lines (may happen in direct mapped and set associative caches).

- *Cache interference misses*: when multiple jobs operate concurrently in a shared cache system and the sum of the WSS of each job exceeds the total cache size, frequent capacity and conflict misses may arise. Jobs that incur frequent level-X capacity and confict misses even if executing in isolation are said to be *trashing* the level-X cache.

Some others terms will be useful. *Cache affinity* describes the fact that a job's overall cache miss rate tends to decrease with the passage of execution time; unless it trashed all cache leves, after an initial burst of misses, the general trend will be little or no other events, since all the useful cache lines will be loaded into the cache. Preemptions or migrations may cause cache affinity to be lost completely for some levels of cache, causing additional compulsory misses. Moreover, a job's memory references are *cache-warm* after cache affinity has been established; on the contrary, *cache-cold* references happen when there is no cache affinity.

1.5 SCHED_DEADLINE implementation

`SCHED_DEADLINE` [38] is a new scheduling policy (made by Dario Faggioli and Michael Trimarchi), implemented inside its own scheduling class, aiming at introducing deadline scheduling for Linux tasks. It is being developed by

Evidence S.r.l. [9] in the context of the EU-Funded project ACTORS [10].
The need of an EDF scheduler in Linux has been already highlighted in the `Documentation/scheduler/sched-rt-group.txt` file, which says: *"The next project will be SCHED_EDF (Earliest Deadline First scheduling) to bring full deadline scheduling to the linux kernel"*. Developers have actually chosen the name `SCHED_DEADLINE` instead of `SCHED_EDF` because EDF is not the only deadline algorithm and, in the future, it may be desiderable to switch to a different algorithm without forcing applications to change which scheduling class they request.
The partners involved in this project (which include Ericsson Research, Evidence S.r.l., AKAtech) strongly believe that a general-purpose operating system like Linux should provide a standard real-time scheduling policy still allowing to schedule non real-time tasks in the usual way.

The existing scheduling classes (i.e., `sched_fair` and `sched_rt`, see fig. 1.1) perform very well in their own domain of application. However,

- they cannot provide the guarantees a time-sensitive application may require. The point has been analyzed for `SCHED_FIFO` and `SCHED_RR` policies (refer to sec. 1.2.1); using `sched_fair` no concept of timing constraint can be associated to tasks.

- The latency experienced by a task (i.e., the time between two consecutive executions of a task) is not deterministic and cannot be bound, since it highly depends on the number of tasks running in the system at that time.

It has to be emphasized the fact that these issues are particularly critical when running time-sensitive or control applications. Without a real-time scheduler, in fact, it is not possible to make any feasibility study of the system under development, and developers cannot be sure that the timing requirements will be met under *any circumstance*. This prevents the usage of Linux in industrial context.

[9] `http://www.evidence.eu.com`
[10] `http://www.actors-project.eu/`

1.5.1 Main Features

SCHED_DEADLINE [11] implements the Earliest Deadline First algorithm and uses the Constant Bandwidth Server to provide *bandwidth isolation* [12] among tasks. The scheduling policy does not make any restrictive assumption about the characteristics of tasks: it can handle periodic, sporadic or aperiodic tasks.

This new scheduling class has been developed from scratch, without starting from any existing project, taking advantage of the modularity currently offered by the Linux scheduler, so do not be too invasive. The implementation is aligned with the current (at the time of writing) mainstream kernel, and it will be kept lined up with future kernel versions.

SCHED_DEADLINE relies on standard Linux mechanisms (e.g., control groups) to natively support multicore platforms and to provide hierarchical scheduling through a standard API.

1.5.2 Interaction with Existing Policies

The addition of sched_deadline scheduling class to the Linux kernel does not change the behavior of the existing scheduling policies, neither best-effort and real-time ones. However, given the current Linux scheduler architecture, there is some interaction between scheduling classes. In fact, since each class is asked to provide a runnable task in the order they are chained in a linked list, "lower" classes actually run in the idle time of "upper" classes. Where to put the new scheduling class is a key point to obtain the right behavior. Developers chose to place it above the existing real-time and normal scheduling classes, so that deadline scheduling can run at the highest priority, otherwise it cannot ensure that the deadlines will be met.
Figure 1.5 shows the Linux scheduling framework with SCHED_DEADLINE added.

[11] The new kernel/sched_deadline.c file contains the scheduling policy core.

[12] Different tasks cannot interfere with each other, i.e., CBS ensures each task to run for at most its runtime every (relative) deadline length time interval.

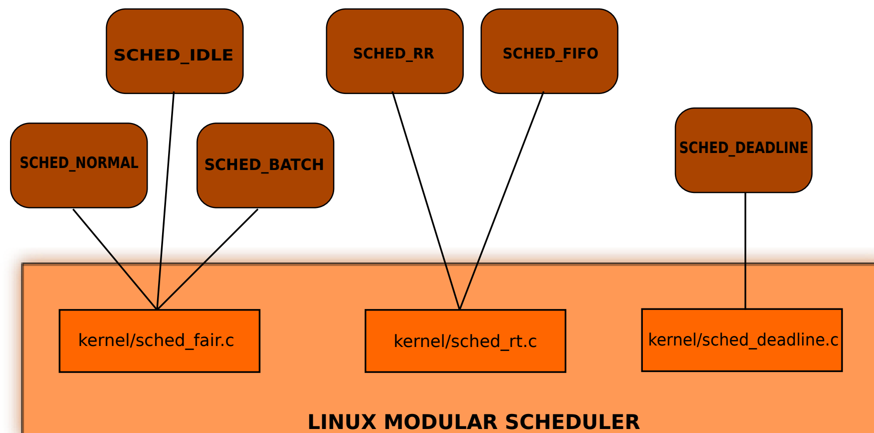


Figure 1.5: The Linux modular scheduling framework with SCHED_DEADLINE.

1.5.3 Current Multiprocessor Scheduling Support

As we have seen in section 1.1.2, inside Linux each CPU has its own ready queue, so the way Linux deals with multiprocessor scheduling is often called *distributed runqueue*. Tasks can, if wanted or needed, migrate between the different queues. It is possible to pin some task on some processor, or set of processors, setting the so called *scheduling affinity* as well.
An opposite solution may be to use a single global runqueue from which each task can be picked up and run on each CPU, but an indepth discussion of advantages and drawbacks of both alternatives is postponed to the next chapter.

SCHED_DEADLINE developers were interested in a general solution, i.e. a globally scheduled system in which it is easy to ask one or more tasks to stay on a pre-specified CPU (or set of CPU). Therefore, the easiest and simplest way of achieving this was to go for the same approach of Linux itself, which means having one runqueue for each CPU implemented with a *red-black tree* [13]. Furthermore, by correctly setting the affinity of each EDF task in the system we can have what the real-time and scheduling community

[13]Ordered by increasing deadline, so it is straightforward to pick the task to run next (at a low computational cost).

calls *partitioned scheduling* [14].

The current implementation totally lacks the possibility of migrate tasks between CPUs. EDF per-CPU runqueues are worlds apart, each task that is born on a CPU dies on the same [15]. We can say that the implementation is *totally partitioned*; then the goal of this thesis is to design the way and develop the code in order to migrate tasks among the runqueues in such a way that:

- we always try to have, on an m CPU system, the m earliest deadline ready tasks running on the CPUs;
- we always respect the affinity the tasks specify.

1.5.4 Task Scheduling

As mentioned earlier, `SCHED_DEADLINE` does not make any restrictive assumption on the characteristics of its tasks, thus it can handle:

- periodic tasks, typical in real-time and control applications;
- aperiodic tasks;
- sporadic tasks (i.e., aperiodic tasks with a *minimum interarrival time* (MIT) between releases), typical in soft real-time and multimedia applications;

A key feature of task scheduling in this scheduling class is that *temporal isolation* is ensured. This means, the temporal behavior of each task (i.e., its ability to meet its deadlines) is not affected by the behavior of any other task in the system. So, even if a task misbehaves, it is not able to exploit larger execution time than it has been allocated to it and monopolize the processor.

Each task is assigned a *budget* (`sched_runtime` and a *period*, considered equal to its *deadline* (`sched_period`). The task is guaranteed to execute for

[14] With *global scheduling* each task may be picked up and run on each CPU.
[15] Unless the user changes the task affinity.

an amount of time equal to `sched_runtime` every `sched_period` (task *utilization* or *bandwidth*). When a task tries to execute more than its *budget* it is slowed down, by stopping it until the time instant of its next deadline. When, at that time, it is made runnable again, its budget is refilled and a new deadline computed for him. This is how the CBS algorithm works, in its hard-reservation configuration.
This way of working goes well for both aperiodic and sporadic tasks, but it imposes some overhead to "standard" periodic tasks. Therefore, the developers have made it possible for periodic tasks to specify, before going to sleep waiting for the next activation, the end of the current instance. This avoid them (if they behave well) being disturbed by the CBS.

1.5.5 Usage and Task API

`SCHED_DEADLINE` users have to specify, before running their real-time application, the system wide `SCHED_DEADLINE` bandwidth. They can do this echoing the desired values in `/proc/sys/kernel/sched_deadline_period_us` and `/proc/sys/kernel/sched_deadline_runtime_us` files. The quantity

$$\frac{\texttt{sched_deadline_runtime_us}}{\texttt{sched_deadline_period_us}}$$

will be the overall system wide bandwidth `SCHED_DEADLINE` tasks are allowed to use.

The existing system call `sched_setscheduler(...)` has not been extended, because of the binary compatibility issues that modifying its `struct sched_param` parameters would have raised for existing applications. Therefore, another system call, called `struct sched_param_ex(...)` has been implemented. It allows to assign or modify the scheduling parameters described above (i.e., `sched_runtime` and `sched_period`) for tasks running with `SCHED_DEADLINE` policy. The system call has the following prototype:

```
struct sched_param_ex {
```

```
    int sched_priority;
    struct timespec sched_runtime;
    struct timespec sched_deadline;
    struct timespec sched_period;
    int sched_flags;
};

int sched_setscheduler_ex(pid_t pid,
      int policy, unsigned int len,
      struct sched_param_ex *param);
```

For the sake of consistency, also

```
int sched_setparam_ex(pid_t pid, unsigned int,
      struct sched_param_ex *param);
int sched_getparam_ex(pid_t pid, unsigned int,
      struct sched_param_ex *param);
```

have been implemented. Another system call,

```
int sched_wait_interval(int flags,
      const struct timespec *rqtp,
      struct timespec *rmtp);
```

allows periodic tasks to sleep till the specified time (i.e., the end of its period).
On multicore platforms, finally, tasks can be moved among different processors using existing Linux mechanisms, i.e. `sched_setaffinity(...)`.

Chapter 2

Design and Development

2.1 EDF scheduling on SMP systems

In this thesis we address the problem of scheduling soft real-time tasks on a Symmetric Multi Processor (SMP) platform, made up by M identical processors (cores) with constant speed. In extending the functionalities of `SCHED_DEADLINE` scheduling class we have to consider the class of soft real-time applications that can be modeled as a set of periodic/sporadic tasks. In soft real-time applications deadlines are not critical, but it is important to respect some kind of *Quality Of Service* (QoS) requirements (e.g., limited number of deadline misses, limited deadline miss percentage, etc. . .).
Applications of interest can tolerate a bounded lateness with respect to the desired deadline. This kind of constraint matches a large class of software in multimedia, telecommunications and finance. As an example, consider a HDTV video streaming or a VoIP session: a given frame-rate must be guaranteed, but a jitter of few milliseconds in the frame-time does not significantly affect the quality of the video. In contrast, audio quality is extremely sensitive to silence gaps. A bounded lateness in providing new samples (deadlines) to the device can be easily compensated using a buffering/play-back strategy.

As previously said `SCHED_DEADLINE` implements the EDF scheduling algorithm, in which, roughly speaking, jobs are scheduled in order of increas-

ing deadlines, with ties broken arbitrarly. The two main approaches to EDF on multiprocessor systems are *partitioned*-EDF (P-EDF) and *global*-EDF (G-EDF).

In P-EDF tasks are statically assigned to processors and those on each processor are scheduled on an EDF basis. Tasks may not migrate. So, in an M processors system we have M task sets independently scheduled. The main advantage of such an approach is its simplicity, as a multiprocessor scheduling problem is reduced to M uniprocessor ones. Furthermore, since there is no migration, this approach presents a low overhead. Drawbacks of P-EDF are: first, finding an optimal assignement of tasks to processors is a bin-packing problem, which is NP-hard (sub-optimal heuristics are usually adopted); second, there are task sets that are schedulable only if tasks are not partitioned [9]; third, when tasks are allowed to dynamically enter and leave the system, a global re-assignement of tasks to processors may be necessary to balance the load, otherwise the overall utilization may decrease dramatically.

In G-EDF jobs are allowed to be preempted and job migration is permitted with no restrictions. Jobs are usually inserted in a global deadline-ordered ready queue, and at each instant of time the available processors are allocated to the nearest deadline jobs in the ready queue.

No variant of EDF is optimal, i.e., deadline misses can occur under each EDF variant in feasible systems (i.e., systems with total utilization at most the number of processors). It has been shown, however, that deadline tardiness under G-EDF is bounded in such systems, which, as we said, is sufficient for many soft real-time applications [14, 42].

A third *hybrid* approach exists (H-EDF [8]) in which tasks are statically assigned to fixed-size clusters, much as tasks are assigned to processors in P-EDF. The G-EDF algorithm is then used to schedule the tasks on each cluster. Tasks may migrate within a cluster, but not across clusters. In other words, each cluster is treated as an independent system for scheduling purposes. Under such an approach, deadline tardiness is bounded for each cluster as long as the total utilization of the tasks assigned to each cluster is at most the number of cores per cluster.

2.2 A look at the Linux SMP support

In the early days of Linux 2.0, SMP support consisted of a *big-lock* that serialized access across the system. While a CPU was choosing a task to dispatch, the runqueue was locked by the CPU, and others had to wait. Advances for support of SMP slowly migrated in, but it wasn't until the 2.6 kernel that full SMP support was developed.
The 2.6 scheduler doesn't use a single lock for scheduling; instead, it has a lock on each runqueue. This allows all CPUs to schedule tasks without contention from other CPUs. In addition, with a runqueue per processor, a task generally shares affinity with a CPU and can better utilize the CPU's hot cache. Another key feature is the ability to load balance work across the available CPUs, while maintaining some affinity for cache efficiency[1].

At the time of writing there are two approaches inside the Linux kernel to to address SMP and load-balance, each contained in its own kernel module. In `kernel/sched_fair.c` we find the CFS implementation. As previously said, the CFS uses a time-ordered red-black tree to enqueue tasks and to build a timeline of future task execution. When tasks are created in an SMP system, they are placed on a given CPU's runqueue. In the general case, you can't know whether a task will be short-lived or it will run for a long time. Therefore, the initial allocation of tasks to CPUs is likely to be suboptimal. To maintain a balanced workload across CPUs, the CFS uses *load balancing*. Periodically a processor[2] checks to see whether the CPU loads are unbalanced; if they are, the processor performs a cross-CPU balancing of tasks.
The `kernel/sched_rt.c` module behaves significantly different for two main reasons. First, since it must deal with task priorities, it uses a priority array to enqueue active tasks. The task to run next is always the highest

[1]When a task is associated with a single CPU, moving it to another CPU requires the cache to be flushed for the task. This increases the latency of the task's memory access until its data is in the cache of the new CPU.

[2]More precisely the *migration thread*, a high priority system thread that performs thread migration.

priority one. Second, stock load balancing functionality has been deactivated. With general processes, which expect high throughput, migration can be done in batch, since no real-time constraint has to be meet. Real-time (RT) tasks must instead be dispatched on a CPU that they can run on as soon as possible. With the CFS approach an RT task can wait several milliseconds before it gets scheduled to run. The migration thread is not fast enough to take care of RT tasks. The RT tasks balancing is done with an active method, i.e., the scheduler actively push or pull RT tasks between runqueues when they are woken up or scheduled. The CFS balancing was removed in order for this to work, or it would actually pull RT tasks from runqueues to which they have been already assigned. Even if an RT task migration means a bit of processor cache inefficiency (line flushing and reloading), it is needed to reduce latency.

2.3 Extending SCHED_DEADLINE

This section is the core of the thesis. We will analyze design choices, that are the foundations, and commenting the code, that is the building structure, of our G(H)-EDF implementation inside the Linux kernel.

2.3.1 Design

In designing a global scheduler there are two things to choose and one to consider: respectively, the nature of the runqueue(s), the method of migration and the need for integration with the existing infrastructure.

Among the various possible alternatives [5, 6], the simplest and most widely used ones are: a global runqueue from which tasks are dispatched to processors, and a distributed approach with one runqueue for CPU and a dynamical allotment of tasks to runqueues.
Advantages of a unique global runqueue are easy implementation and management and no need to face synchronization among the clocks of CPUs. With only one runqueue the scheduler has not to decide where (on which

CPU) to queue a ready task; moreover, assuming a some way ordered runqueue, pick the system-wide task to run next is straightforward.
Unfortunately, the drawbacks win against the benefits of such an approach. In a (large) SMP system scalability is a key goal. This implies the performance of the scheduler on a given system remains the same as one adds more processors to it. With a global runqueue (shared by the CPUs), the performance of the scheduler degraded as the number of processors increased, due to lock contention. The overhead of keeping the scheduler and all of its data structures consistent is reasonably high, i.e., to ensure that only one processor can concurrently manipulate the runqueue, it is protected by a lock. This means, effectively, only one processor can execute the scheduler concurrently. To solve this problem, we usually divide the single global runqueue into a unique runqueue per processor (as in the stock Linux scheduler). This design is often called a multiqueue scheduler. Each processor's runqueue has a separate selection of the runnable tasks on a system. When a specific processor executes the scheduler, it selects only from its runqueue. Consequently, the runqueues receive much less contention, and performance does not degrade as the number of processors in the system increases.
Drawbacks of this case are a complex management of the concurrent runqueues, in terms of tasks' allotment and consistence of the data structures, and the technical difficulty in doing a global scheduling choice (pick the right system-wide task to run next may not be easy, e.g., for the lack of synchronization among the CPUs' clocks).
Considering that, at the present day, the number of CPUs/cores in a machine can even reach the 128 units and that a real-time scheduling decision can not be slowed down by the time to acquire a so probably contended lock, the only viable choice is a multiqueue scheduler (trying to find an efficient solution to manage this approach's complexity).

We recall for section 2.2 that tasks' migration can be active or passive. Similarly to what said for RT tasks, even EDF tasks must be dispatched on a CPU that they can run on as soon as possible. By doing this we can approximate a G-EDF scheduler, where we use the term approximate because

there may be some time intervals in which the scheduler is doing a scheduling decision or a task is beeing migrated and the "on an m CPU system, the m earliest deadline ready tasks running on the CPUs" statement may be violated. Therefore, the approach that we will adopt is to actively push or pull EDF tasks between runqueues when they are woken up or scheduled, deactivating the load balancer.

Integration with the existing framework of the Linux scheduler and a correct use of the data structures already devoted to tasks' migration is compulsory. The Linux scheduler uses (and implements) the concept of *cpu sets* to be able to divide system's CPUs in multiple exclusive sets that can be considered island domains. The notion of a *root-domain* is then used to define per-domain variables. Whenever a new exclusive cpuset is created, also a new `root_domain`[3] is created and attached to it.
The *root-domain* structure contains both the cpu set masks and the variables that control RT tasks migration:

```
struct root_domain {
        atomic_t refcount;
        cpumask_var_t span;
        cpumask_var_t online;

        cpumask_var_t rto_mask;
        atomic_t rto_count;
#ifdef CONFIG_SMP
        struct cpupri cpupri;
#endif
};
```

The `span` and `online` masks identify set and status of the CPUs belonging to a domain; `rto_mask` and `rto_count` allow to know which and how many CPU of the set are *overloaded* (contain more then one RT task); `cpupri` permits efficient management of the system's runqueues. For these reasons, we will work at the `root_domain` level, so that cpu sets still work as expected.

[3] `struct root_domain` is defined in `kernel/sched.c`.

2.3.2 Development

Based on the design choices taken in section 2.3.1, we are now able to dive deep into the code[4]. We will first present the data structures needed to implement the *push and pull* logic, then we will analyze the implementation of that logic with a top-down approach.

Not surprisingly, `struct dl_rq` is the place where we put accounting informations to manage overloading and migrations, together with a tree of *pushable* tasks.

Listing 2.1: `struct dl_rq` extended.

```
struct dl_rq {
  /* runqueue is an rbtree, ordered by deadline */
  struct rb_root rb_root;
  struct rb_node *rb_leftmost;
  unsigned long dl_nr_running;

#ifdef CONFIG_SMP
       struct {
               /* earliest queued deadline task */
         u64 curr;
               u64 next; /* next earliest */
       } earliest_dl;
       unsigned long dl_nr_migratory;
       unsigned long dl_nr_total;
       int overloaded;

       /* pushable tasks RBTREE, ordered by deadline */
       struct rb_root pushable_tasks_root;
       struct rb_node *pushable_tasks_leftmost;
#endif
...
};
```

Struct `earliest_dl` operates like a cache for the earliest deadlines of the

[4] The formatting, indentation and space between lines of the previous and the following pieces of code may be modified from the actual code for a seamless integration with text.

ready SCHED_DEADLINE (DL) tasks on the runqueue, in order to allow an efficient push/pull decision. dl_nr_migratory and dl_nr_total respectively account for the number of DL tasks that can migrate and the total number of queued DL tasks; a flag for the overloaded status is then given by overloaded.

The *pushable task tree* is a red-black tree with its root on pushable_tasks_root. We use an rbtree (sorted by increasing deadlines) because the task to be push/pull next is the one with the earliest deadline. Moreover, the pointer pushable_tasks_leftmost permits an easy handling of the tree.

A slight change in struct root_domain doesn't alter the *cpu sets* architecture and allows a *root domain* level overloading management.

Listing 2.2: struct root_domain extended.

```
struct root_domain {
  atomic_t refcount;
  cpumask_var_t span;
  cpumask_var_t online;
  /*
   * The "RT overload" flag: it gets set if
   * a CPU has more than
   * one runnable RT task.
   */
  cpumask_var_t rto_mask;
  atomic_t rto_count;
#ifdef CONFIG_SMP
  struct cpupri cpupri;
#endif
       /*
        * The "DEADLINE overload" flag: it gets set
  * if a CPU has more than
        * one runnable DEADLINE task.
        */
       cpumask_var_t dlo_mask;
       atomic_t dlo_count;
};
```

Even if the code is quite self-explanatory, dlo_mask shows which CPUs of the system are overloaded and dlo_count keeps count of those.

Starting from the top and then giving a brief overview of the bottom, we first analyze the push and pull functions.

Listing 2.3: *Push* function.

```
static int push_dl_task(struct rq *rq)
{
    struct task_struct *next_task;
    struct rq *furthest_rq;

    if (!rq->dl.overloaded)
        return 0;

    next_task = pick_next_pushable_task_dl(rq);
    if (!next_task)
        return 0;

  retry:
    if (unlikely(next_task == rq->curr)) {
        WARN_ON(1);
        return 0;
    }

    /*
     * It's possible that the next_task slipped in of
     * higher priority than current. If that's the case
     * just reschedule current.
     */
    if (unlikely(dl_time_before(next_task->dl.deadline,
            rq->curr->dl.deadline))) {
        resched_task(rq->curr);
        return 0;
    }

    /* We might release rq lock */
    get_task_struct(next_task);

    /* find_lock_lowest_rq locks the rq if found */
    furthest_rq = find_lock_furthest_rq(next_task, rq);
```

```
if (!furthest_rq) {
    struct task_struct *task;
    /*
     * find_lock_furthest_rq releases rq->lock
     * so it is possible that next_task has migrated.
     *
     * We need to make sure that the task is still on the same
     * run-queue and is also still the next task eligible for
     * pushing.
     */
    task = pick_next_pushable_task_dl(rq);
    if (task_cpu(next_task) == rq->cpu && task == next_task) {
        /*
         * If we get here, the task hasnt moved at all, but
         * it has failed to push. We will not try again,
         * since the other cpus will pull from us when they
         * are ready.
         */
        dequeue_pushable_task_dl(next_task);
        goto out;
    }

    if (!task) {
        /* No more tasks, just exit */
        goto out;
    }

    /*
     * Something has shifted, try again.
     */
    put_task_struct(next_task);
    next_task = task;
    goto retry;
}

deactivate_task(rq, next_task, 0);
set_task_cpu(next_task, furthest_rq->cpu);
activate_task(furthest_rq, next_task, 0);
```

```
    resched_task(furthest_rq->curr);

    double_unlock_balance(rq, furthest_rq);

out:
    put_task_struct(next_task);

    return 1;
}
```

The *push* function first checks the overloaded flag to see if there are DL tasks to push away; then pick from the pushable rbtree the task to try to push next. At this time the `find_lock_furthest_rq` (details below) job is to find and lock a runqueue where the task can immediately run. If found the actual migration is accomplished, else the function just exits or retries.

Listing 2.4: ***Pull*** **function.**

```
static int pull_dl_task(struct rq *this_rq)
{
    int this_cpu = this_rq->cpu, ret = 0, cpu;
    struct task_struct *p;
    struct rq *src_rq;

    if (likely(!dl_overloaded(this_rq)))
        return 0;

    for_each_cpu(cpu, this_rq->rd->dlo_mask) {
        if (this_cpu == cpu)
            continue;

        src_rq = cpu_rq(cpu);

        /*
         * Don't bother taking the src_rq->lock if the
         * next deadline task is known to have further
         * deadline than our current task.
         * This may look racy, but if this value is
         * about to go logically earlier, the src_rq will
         * push this task away.
```

```
 * And if its going logically further,
 * we do not care
 */
if (dl_time_before(this_rq->dl.earliest_dl.curr,
        src_rq->dl.earliest_dl.next))
    continue;

/*
 * We can potentially drop this_rq's lock in
 * double_lock_balance, and another CPU could
 * alter this_rq
 */
double_lock_balance(this_rq, src_rq);

/*
 * Are there still pullable DEADLINE tasks?
 */
if (src_rq->dl.dl_nr_running <= 1)
    goto skip;

p = pick_next_earliest_dl_task(src_rq, this_cpu);

/*
 * Do we have an DEADLINE task that preempts
 * the to-be-scheduled task?
 */
if (p && dl_time_before(p->dl.deadline,
      this_rq->dl.earliest_dl.curr)) {
    WARN_ON(p == src_rq->curr);
    WARN_ON(!p->se.on_rq);

    /*
     * There's a chance that p has an earliest
     * deadline than what's
     * currently running on its cpu. This is
     * just that p is wakeing up and hasn't
     * had a chance to schedule. We only pull
     * p if it has a further deadline than the
     * current task on the run queue
```

```
             */
            if (dl_time_before(p->dl.deadline,
                    src_rq->curr->dl.deadline))
                goto skip;

            ret = 1;

            deactivate_task(src_rq, p, 0);
            set_task_cpu(p, this_cpu);
            activate_task(this_rq, p, 0);
            /*
             * We continue with the search, just in
             * case there's an even earliest deadline task
             * in another runqueue. (low likelyhood
             * but possible)
             */
        }
skip:
        double_unlock_balance(this_rq, src_rq);
    }

    return ret;
}
```

The *pull* function checks all the root domain's overloaded runqueues to see if there is a task that the calling runqueue can take in order to run it (preempting the current running one). If found, this function performs a migration, else continue the search or just exits if there aren't any other runqueues to consider.

The *push* function makes use of the `find_lock_furthest_rq` routine to find (and lock) a suitable runqueue to push a task.

Listing 2.5: *find_lock_furthest_rq* function.

```
static struct rq *find_lock_furthest_rq(struct task_struct *task
    ,
                    struct rq *rq)
{
```

```
struct rq *furthest_rq = NULL;
int tries;
int cpu;

for (tries = 0; tries < DEADLINE_MAX_TRIES; tries++) {
    cpu = find_furthest_rq(task);

    if ((cpu == -1) || (cpu == rq->cpu)) {
        break;
    }

    furthest_rq = cpu_rq(cpu);

    /* if the prio of this runqueue changed, try again */
    if (double_lock_balance(rq, furthest_rq)) {
        /*
         * We had to unlock the run queue. In
         * the mean time, task could have
         * migrated already or had its affinity changed.
         * Also make sure that it wasn't scheduled on its rq.
         */
        if (unlikely(task_rq(task) != rq ||
                !cpumask_test_cpu(furthest_rq->cpu,
                        &task->cpus_allowed) ||
                task_running(rq, task) ||
                !task->se.on_rq)) {

            raw_spin_unlock(&furthest_rq->lock);
            furthest_rq = NULL;
            break;
        }
    }

    /* If this rq is still suitable use it. */
    if (dl_time_before(task->dl.deadline,
            furthest_rq->dl.earliest_dl.curr) ||
            (furthest_rq->dl.dl_nr_running == 0)) {
        break;
    }
```

```
        /* try again */
        double_unlock_balance(rq, furthest_rq);
        furthest_rq = NULL;
    }

    return furthest_rq;
}
```

The `find_lock_furthest_rq` function tries `DEADLINE_MAX_TRIES` times to find a suitable runqueue (where the running task has a further deadline than that of the task at issue). It only acquires a double lock (source and destination runqueues) if it succeeds in its work. A check is performed soon after the lock to see if the destination runqueue is still suitable (something can be changed in the meantime).

The very core of all the mechanism, however, are the `furthest_cpu_find` and `find_furthest_rq` functions. The first tries to find the best CPUs in the span (and builds a mask of those) for the task passed as an argument.

Listing 2.6: ***furthest_cpu_find* function.**

```
static int furthest_cpu_find(const struct cpumask *span,
         const struct task_struct *task,
         struct cpumask *furthest_mask)
{
    int cpu;
    int found = 0;
    struct rq *rq;
    struct dl_rq *dl_rq;
    const struct sched_dl_entity *dl_se = &task->dl;
    for_each_cpu(cpu, span) {
        rq = cpu_rq(cpu);
        dl_rq = &rq->dl;
        if ((dl_time_before(dl_se->deadline,
                dl_rq->earliest_dl.curr) ||
                (dl_rq->dl_nr_running == 0)) &&
                cpumask_test_cpu(cpu, &task->cpus_allowed)) {
            cpumask_set_cpu(cpu, furthest_mask);
            found = 1;
```

```
        } else {
            cpumask_clear_cpu(cpu, furthest_mask);
        }
    }
    return found;
}
```

This function is then utilized by `find_furthes_rq` to elect the best CPU based on task's affinity and system's topology.

Listing 2.7: ***find_furthest_rq*** **function.**

```
static int find_furthest_rq(struct task_struct *task)
{
    struct sched_domain *sd;
    struct cpumask *furthest_mask = __get_cpu_var(local_cpu_mask)
        ;
    int this_cpu = smp_processor_id();
    int cpu       = task_cpu(task);

    if (task->dl.nr_cpus_allowed == 1)
        return -1; /* No other targets possible */

    if (!furthest_cpu_find(task_rq(task)->rd->span, task,
                furthest_mask)) {
        return -1; /* No targets found */
    }

    /*
     * At this point we have built a mask of cpus
     * representing the furthest deadline tasks in the system.
     * Now we want to elect
     * the best one based on our affinity and topology.
     *
     * We prioritize the last cpu that the task executed on since
     * it is most likely cache-hot in that location.
     */
    if (cpumask_test_cpu(cpu, furthest_mask)) {
        return cpu;
    }
```

```
/*
 * Otherwise, we consult the sched_domains span maps
 * to figure out which cpu is logically closest to our
 * hot cache data.
 */
if (!cpumask_test_cpu(this_cpu, furthest_mask))
   this_cpu = -1; /* Skip this_cpu opt if the same */

for_each_domain(cpu, sd) {
   if (sd->flags & SD_WAKE_AFFINE) {
      int best_cpu;

      /*
       * "this_cpu" is cheaper to preempt than a
       * remote processor.
       */
      if (this_cpu != -1 &&
          cpumask_test_cpu(this_cpu,
         sched_domain_span(sd)))
         return this_cpu;

      best_cpu = cpumask_first_and(furthest_mask,
                 sched_domain_span(sd));

      if (best_cpu < nr_cpu_ids)
         return best_cpu;
   }
}

/*
 * And finally, if there were no matches within
 * the domains just give the caller *something*
 * to work with from the compatible
 * locations.
 */
if (this_cpu != -1)
   return this_cpu;

cpu = cpumask_any(furthest_mask);
```

```
    if (cpu < nr_cpu_ids)
        return cpu;
    return -1;
}
```

At this point we have seen how a migration is performed, both for the pull and the push case. However, we lack to understand how the overleaded status is managed and where in the code a push/pull call may arise.

Three functions are responsible for the overloaded status consistency, and they are quite self-explanatory. Other two control the correct update of `dl_nr_running` and `earliest_dl`, but are not worth to be listed here to avoid too much confusion.

Listing 2.8: *Overloaded status* management functions.

```
static void update_dl_migration(struct dl_rq *dl_rq)
{
    if (dl_rq->dl_nr_migratory && dl_rq->dl_nr_total > 1) {
        if (!dl_rq->overloaded) {
            dl_set_overload(rq_of_dl_rq(dl_rq));
            dl_rq->overloaded = 1;
        }
    } else if (dl_rq->overloaded) {
        dl_clear_overload(rq_of_dl_rq(dl_rq));
        dl_rq->overloaded = 0;
    }
}

static void inc_dl_migration(struct sched_dl_entity *dl_se,
            struct dl_rq *dl_rq)
{
    dl_rq = &rq_of_dl_rq(dl_rq)->dl;

    dl_rq->dl_nr_total++;
    if (dl_se->nr_cpus_allowed > 1)
        dl_rq->dl_nr_migratory++;

    update_dl_migration(dl_rq);
}
```

```
static void dec_dl_migration(struct sched_dl_entity *dl_se,
            struct dl_rq *dl_rq)
{
    dl_rq = &rq_of_dl_rq(dl_rq)->dl;

    dl_rq->dl_nr_total--;
    if (dl_se->nr_cpus_allowed > 1)
        dl_rq->dl_nr_migratory--;

    update_dl_migration(dl_rq);
}
```

The functions listed in 2.8 are called every time a DL entity is queued in or dequeued from a CPU's runqueue.

Finally, four points in the code can lead to a push/pull mechanism activation. The push function is called after each scheduling decision (inside `post_schedule_dl`) and every time a DL task is woken up (inside `task_woken_dl`), so to provide a sort of load balancing. The pull function is instead called before each scheduling decision (inside `pre_schedule_dl`) and every time the last DL task on a runqueue ends its execution (inside `switched_ from_dl`) and consequently leaves room for the others DL tasks of the system.

Chapter 3

Experimental Results

3.1 Cases of Interest

Decide what to measure and how to collect data from a scheduling experiment is not a straightforward task. Moreover, a scheduling algorithm performance analysis may be influenced by a number of subtle events that affect how the system behaves, introducing unexpected noise in the collected data.
The target of the following analysis is not to prove that our G-EDF implementation has a very good overall performance or that it behaves better than other real-time algorithms for some specific application. Although this kind of analysis may be very interesting, also to understand which type of applications can find advantages in the use of a global EDF scheduler, it is beyond the scope of this thesis. We will focus instead on evaluate system's overheads introduced by the migration of tasks. It is very important to verify that our implementation can be a viable choiche for soft real-time applications developers, providing them a new useful mechanism inside the Linux kernel.

3.1.1 Working Set Size

The focus of a large part of this analysis is how the presence of cache memory influences the behaviour of the implemented algorithm, considering the introduced overhead.

Since cache memory is very fast, compared to the other system's components, we first need to find a working set size that allows us to make measurements of cache-related operations only. As will be clear in the next few paragraphs, working with a too small dataset, we may incurr in measuraments errors due to the fact that the time spent on the cache is comparable to the time spent on executing code.

To find a suitable working set size, we conducted several experiments with successive runs of a purpose built application. As suggested in a technical article by Ulrich Drepper [17], we wrote a program which can simulate working sets of arbitrary size, read and write access, and sequential or random access. The program creates an array corresponding to the working set size of elements of this type:

```
struct list_elem {
        volatile struct list_elem *next;
        long int pad[NPAD];
};
typedef struct list_elem item;
```

All entries are chained in a circular list using the n element, in sequential order. Advancing from one entry to the next always uses the pointer, even if the elements are laid out sequentially. The `pad` elements is the payload and it can be varied as needed. A working set of 2^N bytes contains 2^N`/sizeof(struct list_item)` elements. Obviously, `sizeof(struct list_elem)` depends on the value of `NPAD`. For 64-bit systems, as those we have used, `NPAD` $= 3$ means the size of each array element is 32 bytes and `NPAD` $= 7$ means 64 bytes.

The program allocates a static `work_set` array of `item` elements, increasing the working set size by a power of 2 at every iteration. Each iteration works as follows:

- locks the working set's memory locations and builds the circular list;
- for `MAX_ITER` times:
    - flushes the cache,

- accesses all the elements of the array,
- accesses, a second time, a fixed number of the elements (with a write to read ratio of 1/4);

- unlocks the memory.

Measures, on terms of execution time and using performance counters, are taken for the first and the second access to the vector. This is done in order to simulate a *cache-cold* and a *cache-warm* access to the same working set, in fact, after the cache flush no elements of the array are present in the cache; then, with the first access, the program loads every element in the cache, so generating compulsory level-1 and level-2 cache misses; the second time it accesses the vector, and considering that the vector is smaller in size that the level-1 cache size and little or no interference with other tasks of the system, the program should find already on the cache everything he needs, spending less time and generating less or no cache misses in doing its job.

It has to be said the we used performace counters in order to prove the expected program behaviour as well, in this respect we also analyzed working set sizes that exceed L1D and L2 cache sizes.

3.1.2 Cache-Related Overhead

Once we have restricted our analysis to one or two working sets (considered interesting in relation with the hardware we used), we have now to develop a general method of "cache overhead accounting", in order to prove that the overhead introduced by migrations is negligible and doesn't affect so much the overall system's behaviour.

The first step in finding cache-related overheads is to build a synthetic worst case situation and look at how the system behaves. To heavy stress cache memory we built an application that runs from 1 to 30 concurrent tasks using a FIFO scheduling mechanism. Each tasks builds the circular list according to a specified WSS; then, until a shutdown signal, repeatedly accesses a fixed number of elements of the list (`ACCESSED_ITEMS`) measuring, at each iteration, access time, L1D and L2 cache misses. Moreover,

every time a task finishes an iteration, it yields the processor; since tasks are scheduled with a FIFO approach and they all are at the same priority, when a task reach the end of an iteration, it will access again to its list only after all the others tasks.
This way we recreate a worst case situation, as every time a task wants to access to its data, he certainly doesn't find them on the cache because they were all flushed away by the others tasks.

Worst case assumption will be confirmed looking at L1D and L2 cache misses. Since in this thesis we focus on inter-core migration, we expect a large amount of L1D cache misses (the sum of all the tasks' WSSes will be less than the L2 cache size) and little or no L2 misses. With access time measures we will instead make an estimation of the worst case per-item access time, dividing total access time by the fixed number of accesses.

We are now able to deal with periodic tasks. A program has been built that reads from a configuration file a task set and creates periodic tasks that execute for a specified amount of time. For each task is possible to specify task's execution time and period, then the program calculates how many accesses to the list a task must do to reach its execution time simply by dividing that value for the estimated worst-case access time. After the calculated number of access a task sleeps until the end of its period. Having instrumented the code, we are able to collect data about real execution time (and percentual error compared to the requested execution time), lateness in waking up a task after its new activation time, slack or tardiness (the amount of time the task finish its job until the end of its period or how much it is gone beyond that instant of time), L1D and L2 cache misses.

The use of those kind of measures is fundamental in order to understand the amount of cache-related overhead a scheduling mechanism introduces allowing tasks to migrate between processors. The idea behind this statement is that, if we schedule a schedulable task set under some scheduling algorithm that allows tasks migrations and we find that every tasks in the system is able to execute for the requested amount of time and, at the same time, doesn't miss its deadlines (assuming deadlines coincident with period), than we can state that the cache-related overhead introduced by that algorithm is

negligible, else the algorithm operates in a so poor way that it is not usable in a real system.

3.2 Software Tools

Care must be taken when trying to collect data from a running system. The collecting tools must introduce little or no overhead in order to not influence measurement. Luckily, some software tools exist both in kernel and user space to address this kind of problem. In this section we will present the tools we used to take measures, with simple snippets of code to clarify how they work.

3.2.1 Tracing the kernel

There are quite a variety of tracing options for Linux, we have decided to use *ftrace* for its low overhead in tracing and its smooth integration with user space applications. The name *ftrace* comes from "function tracer", which was its original purpose, but it can do more than that. Various additional tracers have been added to look at things like context switches, how long interrupts are disabled, how long it takes for high-priority tasks to run after they have been woken up, and so on.

First thing to do to use this tracer is enable the proper kernel config option `CONFIG_FUNCTION_TRACER`. Then we can access ftrace through the debug file system, typically mounted this way:

```
# mount -t debugfs nodev /debug
```

That creates a `/debug/tracing` subdirectory which is used to control ftrace and for getting output from the tool. The list of the available tracers is simply showed by reading `/debug/tracing/available_tracers`. Then, one can choose a particular trace writing its name in `current_tracer` [1].

[1] From here on, we assume we are inside the tracing directory.

Tracing is enabled by default, so we have to stop it until the execution of the activity to be traced.

```
# echo 0 > tracing_on
```

Ftrace stores lines of trace in a circular buffer. The execution for a large amount of time can cause buffer overflow (not an error, but early data may be unwittingly discarded). With:

```
# echo 10240 > buffer_size_kb
```

we can control the buffer size in units of kilobytes. Before tracing it may be useful to empty the trace file with:

```
# head -4 trace > trace
```

Then a simple ftrace run may be:

```
# echo 1 > tracing_on
  ...some commands or activity to trace...
# echo 0 > tracing_on
```

Function tracer (`function`) is the one we used to trace the execution of the scheduler. Other than restrict the tracing to `push_dl_task` and `pull_dl_task` functions, we have introduced some *tracepoints* directly inside the kernel. *Tracepoints* [13] are lightweight hooks that can be put at important locations in the kernel code. When the code execution reaches a tracepoint an event is issued that can be trapped by the function tracer. First of all a tracepoint must be defined in `include/trace/events/sched.h` inside the kernel source directory. Follows an example of a tracepoint definition:

Listing 3.1: Tracepoint definition.

```
TRACE_EVENT(push_dl_entry,
        TP_PROTO(struct task_struct *curr),
        TP_ARGS(curr),
        TP_STRUCT__entry(
                __field(        pid_t, pid              )
```

```
                __field(        u64, deadline           )
        ),
        TP_fast_assign(
                __entry->pid            = curr->pid;
                __entry->deadline       = curr->dl.deadline;
        ),
        TP_printk("PID=%d DEADLINE=%llu",
                  __entry->pid, __entry->deadline)
);
```

As we can see a tracepoint can take some arguments and can also print directly on the trace. After the definition, the code can be instrumented with a tracepoint call:

Listing 3.2: Tracepoint call.

```
static int push_dl_task(struct rq *rq)
{
        struct task_struct *next_task;
        struct rq *furthest_rq;

        if (!rq->dl.overloaded)
                return 0;

        trace_push_dl_entry(rq->curr);

        next_task = pick_next_pushable_task_dl(rq);
        if (!next_task) {
                trace_push_dl_exit(rq->curr);
                return 0;
        }
  ...
```

In the previous code snippet we can see how we instrumented our G-EDF implementation for tracing purposes. Finally, at run time, an event must be set to link the tracepoint with the tracing environment:

```
# echo push_dl_entry >> set_event
```

The `set_event` file contains a list of events that must be traced. Care must be taken on echoing new events on it, they must be appended or they will

overwrite previously added ones.

3.2.2 High resolution clocks

One of the common ways to measure time intervals in a user space application is the use of high resolution clocks. Linux, by the `clock_gettime(...)` function, allows to read all the available system's clock. The `clock_gettime (clockid_t clk_id, struct timespec *tp)` function retrives the time of a specified clock `clk_id` and writes it in the `timespec` structure passed as a pointer in the function call.
The `clk_id` argument is the identifier of the particular clock on which to act. A clock may be system-wide and hence visible for all processes, or per-process if it measures time only within a single process. Sufficiently recent versions of glibc and the Linux kernel support the following clocks:

- `CLOCK_MONOTONIC`: clock that represents monotonic time since some unspecified starting point. Very useful to implement periodic tasks (see below).
- `CLOCK_THREAD_CPUTIME_ID`: thread-specific CPU-time clock. Every thread has is on clock of this type, it is used in measuring time intervals with no regards for other tasks concurrent execution.

The `timespec` structure is specified in `<time.h>` and has the following shape:

```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
};
```

We have two uses of high resolution clocks in collecting data for our analysis: measure of time intervals and implementation of periodic tasks.
The common way of measure a time interval inside a thread is the following:

```
clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t_start);
```

```
...some computations to measure...
clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t_stop);
```

Then the difference between `t_stop` and `t_start` gives the execution time of the measured computation. It is to be noted here that this time interval comprises the thread-specific time only, if a thread is preempted and than resumes execution, that time interval is not computed.

The simplest way of doing a periodical activity follows:

```
clock_gettime(CLOCK_MONOTONIC, &t_sleep);
while(1){
    ...some computations here...
    t_sleep = timespec_add(&t_sleep, &t_period);
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                    &t_sleep, NULL);
}
```

In a few words, a time reference is taken the first time a periodic task starts its execution; then the task enters an endless loop where: makes some computations (periodic job), adds to the time reference its period, then sleeps until its next activation time (`clock_nanosleep(...)` does this job).

3.2.3 Performance Counters

Hardware performance counters, or *performance counters*, are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems. With those counters we are able to conduct low-level performance analysis and inspect what's going on deep into the system. Compared to software profilers, performance counters provide low-overhead access to great number of detailed performance information related to CPU's functional units, caches and main memory.

Accessing to performance counters is straightforward, correlating the low level performance metrics back to the source code is the hard task. There are two methods on Linux to work with them, an indirect and a direct one. To the first type belong, most commonly used, OProfile [31] and Performance

Counters for Linux (perf). Even if OProfile may have a larger number of features in comparison with perf, it is not always easy to use and is something external to the Linux kernel. Performance Counters for Linux is instead a subsystem of the kernel devoted to hardware monitoring. It is quite new (introduced with the 2.6.31 kernel), but works flawlessly and integrates well with Linux. The perf utility can be used only for data collection outside the code of the monitored application. If, for example, we want to know how many cache misses an application may caused, we can do this simply issuing the following:

```
perf stat -e L1-dcache-load-misses ./myApp
```

The `perf stat` will run the specified application and gather performance counter statistics of it. With the `-e` option we can specify at which type of events we are interested.

The problem here is how to correlate a specific piece of code with application-wide statistics. A library has been developed that can help us, PAPI [32]. PAPI works on top of the perf subsystems and behaves very similar to how high resoluiton clocks are used.
This library requires a suitable configured kernel, i.e. one with `CONFIG_PERF_COUNTERS=y`. After the library installation (refer to the on-line documentation) the list of availabe hardware events and hardware informations can be showed with `papi_avail`.

```
$ papi_avail

Available events and hardware information.
-----------------------------------------------------
PAPI Version              : 4.0.0.3
Vendor string and code    : GenuineIntel (1)
Model string and code     : Intel(R) Core(TM)2 Duo CPU
                            T7300  @ 2.00GHz (15)
...
Hdw Threads per core      : 1
Cores per Socket          : 2
```

```
...
Total CPU's              : 2
Number Hardware Counters : 5
...
-----------------------------------------------------
The following correspond to fields in the
PAPI_event_info_t structure.

    Name         Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No    Level 1 data cache
                                     misses
PAPI_L1_ICM  0x80000001  Yes   No    Level 1 instruction
                                     cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes   Level 2 data cache
                                     misses
PAPI_L2_ICM  0x80000003  Yes   No    Level 2 instruction
                                     cache misses
```

Moreover, another useful utility may be `papi_mem_info`, that reports detailed system's cache memory infos such as total size of each level, line size, associativity and number of lines.

```
$ papi_mem_info

Memory Cache and TLB Hierarchy Information.
-----------------------------------------------------
...

Cache Information.

L1 Data Cache:
  Total size:           32 KB
  Line size:            64 B
```

```
  Number of Lines:      512
  Associativity:          8

L1 Instruction Cache:
  Total size:            32 KB
  Line size:             64 B
  Number of Lines:      512
  Associativity:          8

L2 Unified Cache:
  Total size:          4096 KB
  Line size:             64 B
  Number of Lines:    65536
  Associativity:         16
```

The library use is then quite easy, first of all the library header must be included in the source code, and it is also convenient to define the number of events we will inspect:

```
#include <papi.h>
...
#define NUM_EVENTS 2
```

Then we must issue an init call in the program's main:

Listing 3.3: PAPI init call.

```c
int main(int argc, char **argv)
{
  ...
  int Events[NUM_EVENTS] = {PAPI_L1_DCM,
          PAPI_L2_TCM};
  long long values[NUM_EVENTS];

  ...

  /* Initialize PAPI library */
```

```
        retval = PAPI_library_init(PAPI_VER_CURRENT);
        if (retval != PAPI_VER_CURRENT) {
                fprintf(stderr,
            "PAPI library init error!\n");
                exit(1);
        }
```

Now is time to start counters, this can be done at the very beginning of the code since succesive reads will reset the counter's value.

Listing 3.4: PAPI counters start.

```
if ((retval = PAPI_start_counters(Events, NUM_EVENTS))
                        != PAPI_OK) {
                fprintf(stderr,
    "PAPI library start counters error!\n");
                if (retval == PAPI_ECNFLCT)
                        fprintf(stderr,
      "Non compatible event set!\n");
                exit(1);
        }
```

After this instant of time successive calls to a read function will return the needed values and reset the counters. An example of code sampling follows:

Listing 3.5: PAPI sampling.

```
if ((retval = PAPI_read_counters(values, NUM_EVENTS))
                                        != PAPI_OK) {
  fprintf(stderr,
  "PAPI library read counters error!\n");
        exit(1);
}

...some computation here...

if ((retval = PAPI_read_counters(values, NUM_EVENTS))
                                        != PAPI_OK) {
  fprintf(stderr,
  "PAPI library read counters error!\n");
        exit(1);
}
```

An access on the `values` vector now gives us the collected data. Last thing to do is to stop the counters.

Listing 3.6: PAPI stop counters.

```
if ((retval = PAPI_stop_counters(values, NUM_EVENTS))
                        != PAPI_OK) {
        fprintf(stderr,
  "PAPI library stop counters error!\n");
  exit(1);
}
```

PAPI library works well in a multithreaded application too. The multithread support must be initialized in the main:

Listing 3.7: PAPI multithread init.

```
/* Initialize PAPI threads support */
retval = PAPI_thread_init(pthread_self);
if (retval != PAPI_OK) {
  fprintf(stderr, "PAPI threads init error!\n");
        exit(1);
}
```

Then start, read and stop of counters goes in the thread body.

3.2.4 Tasks Sets Generator

A generator of random tasks sets is needed in order to not bias experiments. Once a tasks set is generated it must pass a schedulability test, this is strongly advised as to get meaningful measures of the system's behaviour. To that purpose, we extended a tasks sets generator made by G. Lipari and in this section we briefly explain how it works.

The program can take several arguments as input (e.d., the number of tasks that will be part of the tasks set, the number of CPUs on the system, the number of tasks sets to be generated, etc...) and it returns as output a file for each tasks set that passed the schedulability tests. It is possible to specify the number of tasks sets to generate as well, and the generator continues its work until it reaches the desired amount.

The core of this program (the actual creation of tasks) works as follows:

- extracts a random number of tasks between the max and min values passed as arguments;
- calculate the total bandwidth assigned to the tasks set relating it to the number of processor, this value is comprised between a max and a min as well;
- assigns to each task an equal fraction of the total bandwitdh;
- uses the *concavity* (passed as argument) as to shuffle tasks' utilization, an example of this is given in Figure 3.1.

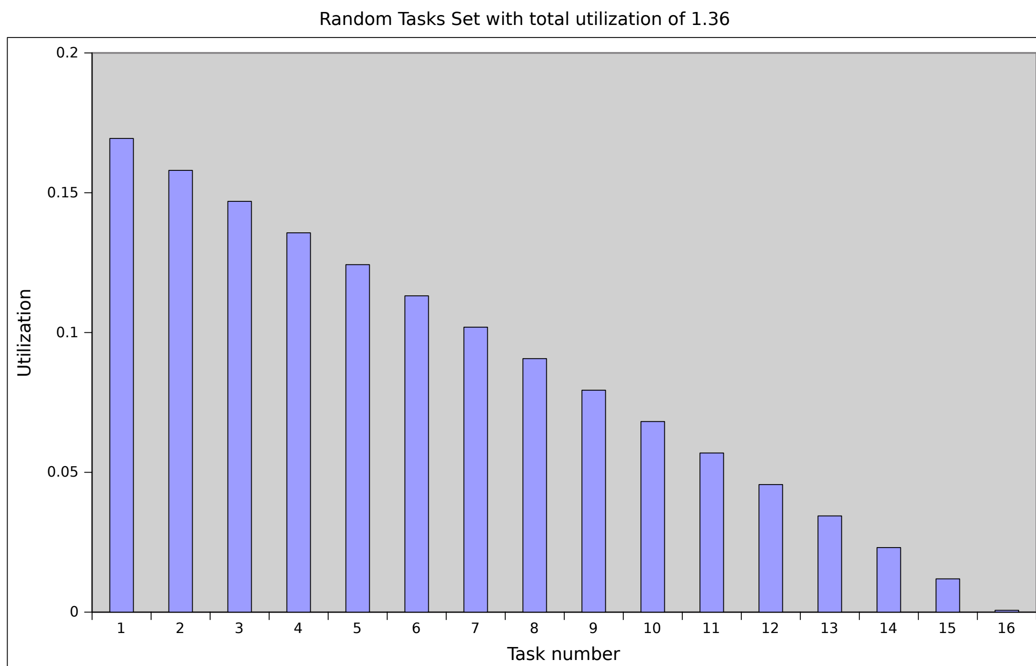


Figure 3.1: An example of a random tasks set.

After a tasks set has been created it must pass every schedulability test chosen by the user in order to be part of the files set generated as output. What follows better clarifies how the program behaves after a tasks set generation.

- If P-EDF is set, the generator tries to divide the tasks among the CPUs with a firts-fit approach. For every task it controls if its utilization can fit inside a CPU's maximum allowed bandwidth (`Umax`). If it can, the program goes to the next task, else tries until it doesn't find a suitable CPU. If no CPU is found the routine just exits and the task set is discarded. On the contrary, if successful, every tasks in the tasks set will have its affinity properly set.
- If G-EDF is set, the tasks set will be tested with three schedulability tests: RTA_EDF, Sanjoy Baruah's RTSS and the Marko Bertogna's improved Sanjoy's test. If a tasks set passes one of these tests, it is considered schedulable and the actual file is created.
- If P-RM is set, the program tries to divide the tasks among the CPUs, as in the P-EDF case, with the difference that every "partial" partitioned tasks set must pass the RTA_FP test. If the adding of a task on each CPU causes the test to fail the tasks set is discarded. It is to be said that, if a tasks set has been previously partitioned with P-EDF, each partition must pass the RTA_FP test to the output file to be created.
- The G-RM case is similar to the G-EDF one, changes the test only, as for RM we use the RTA_FP test.

3.3 Hardware

We used one hardware platform to conduct the experiments:

- an Intel^R Core^{TM}2 Duo T7300 dual-core machine with 32KB of L1D (plus 32KB of L1I) per-core cache memory and 4096KB of L2 shared cache memory.

The use of an Intel^R Core^{TM}2 Quad Q6600 quad-core machine with 32KB of L1D (plus 32KB of L1I) per-core cache memory and 4096KB of L2 shared cache memory is scheduled in the future as to continue the analysis of the

present thesis. Although the first machine was useful to cache related overhead experiments, the second will be necessary for analysing the scheduling mechanism behaviour in a more general context from the point of view of migrations overhead and counting.

3.4 Results

If in section 3.1 we have presented the theoretical aspects of our experimental analysis, here we will show the results of the experiments led on real hardware.

3.4.1 Working Set Size Selection

Working set size selection has been done using the `sinThSeqAccFixed.c` program (for source code see section A.2). The Intel^R Core^{TM}2 Duo T7300 processor has 64 bytes wide cache lines both for level-1 and level-2 caches. So, we have decided to consider the size of each element of the list to be 64 bytes or 32 bytes long. This is done in order to verify the cache behaviour in regard to the number of cache misses.

As said in section 3.1.1, we start the experiment with a working set size of 1KB and we step up it to reach the size of 8192KB (8MB), it is to remember that the T7300 has a 32KB L1D and a 4096KB L2 cache memories.

In Figure 3.2 on the next page is shown the 64 bytes long case. The blue line represents the average per-item access time (in μs); the fuchsia and yellow lines respectively depict L1D and L2 cache misses. The first two measurements are polluted by noise. The measured workload is simply too small to filter the effects of the rest of the system out. We can safely assume that the values are all under $0,018\mu s$.

With this in mind we can see three distinct levels:

- up to a working set size of 16KB,
- from 32KB to 2048KB,

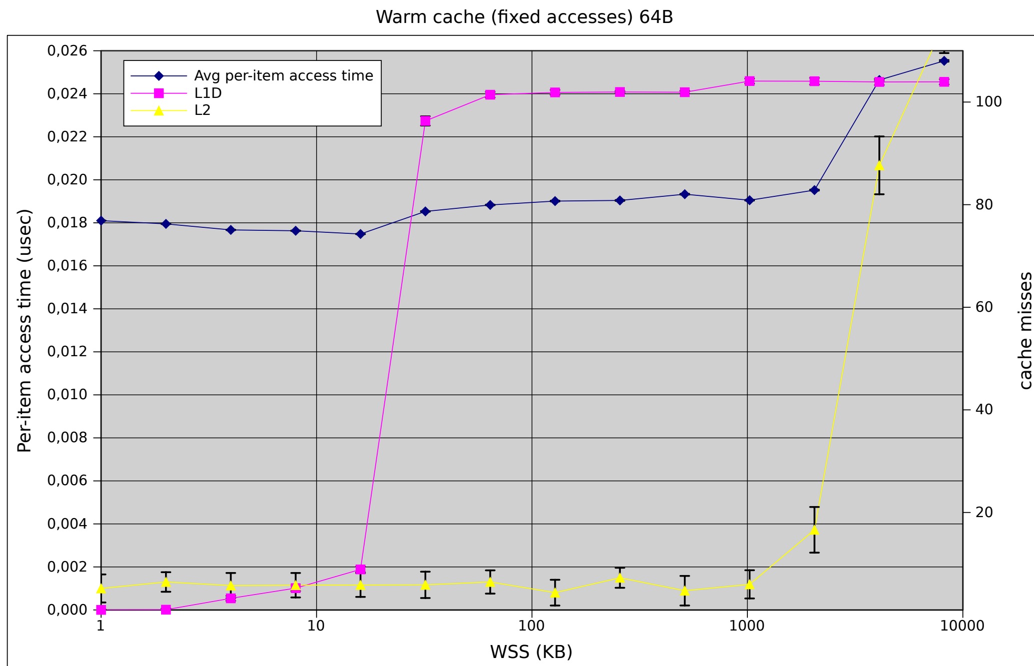


Figure 3.2: Single thread sequential access with elements of 64 bytes.

- from 4096KB and up.

We do not see sharp edges in the transition from one level to the other because the caches are used by other parts of the system as well and so the cache is not exlusively available for the program data. Moreover, the L2 cache is a unified cache also used for instructions.
The trend is confirmed looking at L1D and L2 cache misses and the explanation for it follows:

- up to half of the L1D cache size a cache warm access doesn't raise any cache miss, so it is very fast to access data that are already present in the nearest cache level;

- when the working set size goes beyond 16KB (and considering the aforementioned interference) accesses to the list start to be *L1D cache warm*, in the sense that needed data start to be evicted from L1D for space reasons; per-item access time continues to be slightly above $0,018\mu s$ since L2 cache is warm and loading data from it is faster than a main

memory access;

- when, at last, the working set size reach the L2 cache size we find an abrupt increase of per-time access time due to the fact that, every time we want to access an element of the list, we don't find it on the cache because it was evicted by previous accesses; from this time on we will experiment capacity cache misses.

The other case (elements of 32 bytes) is shown in Figure 3.3. As we can see the trend is the same of the previous case, specifically for the number of cache misses. Per-item access time seems instead to be a little disturbed by system's interfecence, this may depend by the fact that elements are now half a cache line big (generating half the number of cache misses), so the relative number of "interferencing" cache misses is bigger.

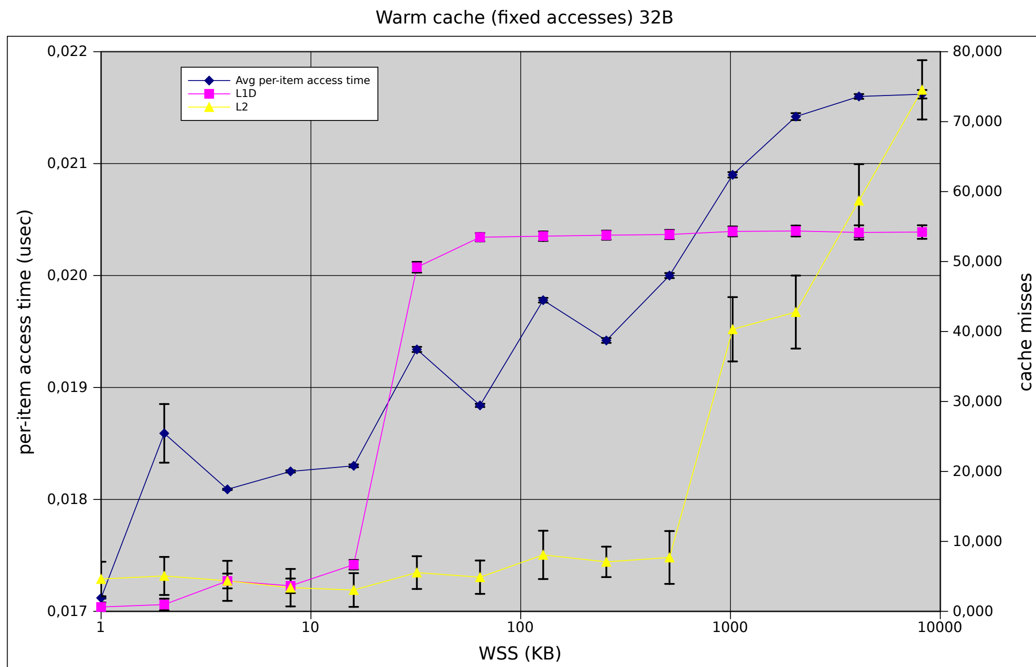


Figure 3.3: Single thread sequential access with elements of 32 bytes.

To mitigate system's interference we have decided to choose 64 bytes as the size of each elements. Another important thing to remark is the fact that the analysis of a syntetic case, for which we know a priori how the trend

must be, and the adherence of the collected data to that trend, proves the correctness of both the analysis method and the way of collecting data.

To further test the goodness of this kind of analysis we conducted another experiment. This time the list is scanned in a random way, we can decide the value of `MAX_HOP` from the current element to the next we can make in doing a sort of *random walk*.

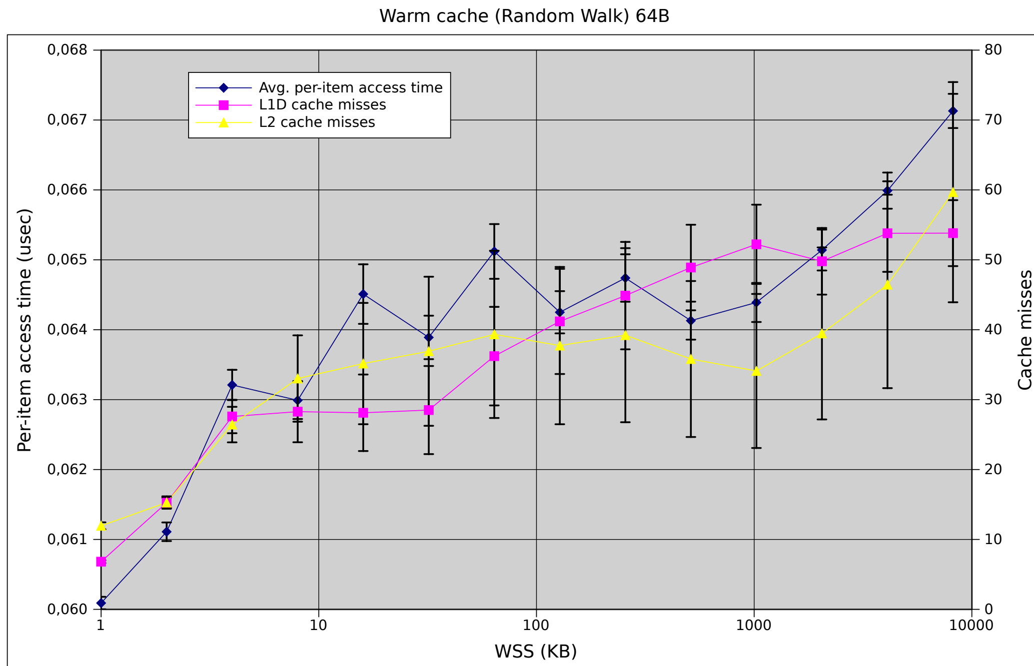


Figure 3.4: Single thread *random walk* with elements of 64 bytes.

In Figure 3.4 we can see that the trend previously depicted nearly vanishes. The fact that the cache is warm dosen't lower so much neither the per-item access time nor the number of cache misses. The reason why is that the processor can't efficientely prefetch random data and you never can tell that the elements accessed the first run will be the same the second time (so invalidating the *cache warm* assumption).

In order to analyse a smooth and interference-free application we will focus our attention, for the following experiments, to a 16KB working set size with a sequential access to the elements of the list.

3.4.2 Cache-Related Overhead

Worst case per-item access time estimate has been done with the use of `multipleThSeqAcc.c` program (for source code see section A.2). The experiment has been run with an increasing number of threads to try to reach a "saturation point". We expect that when the sum of the working sets will exceed half the size of the L1D cache, an abrupt increase on the number of cache misses and a consequent boost on the per-item access time will occur.

The expected behaviour is shown in Figure 3.5. From this graph we can see three important things:

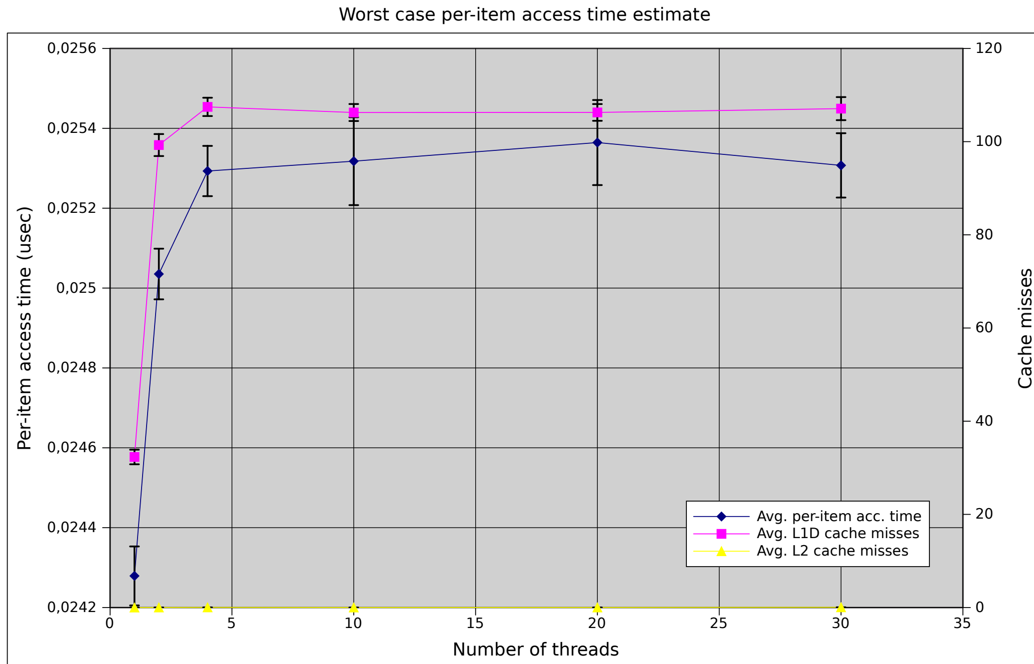


Figure 3.5: Worst case per-item access time estimate (16KB).

- L1D cache misses are nearly absent when there are few threads; using a per-thread working set of 16KB, two threads are enough to exceed the L1D cache size, so reaching the saturation point.
- L2 cache missess are stuck to zero as the number of threads increases; this is the right behaviour since $16 \cdot 30 = 480$(KB), so L2 cache doesn't

experiment capacity misses.

- per-item access time saturates at the rounded down value of $0.025\mu s$.

As to confirm our beliefs we repeated the experiment with a per-thread working set size of 400KB. Figure 3.6 shows how the system behaves.

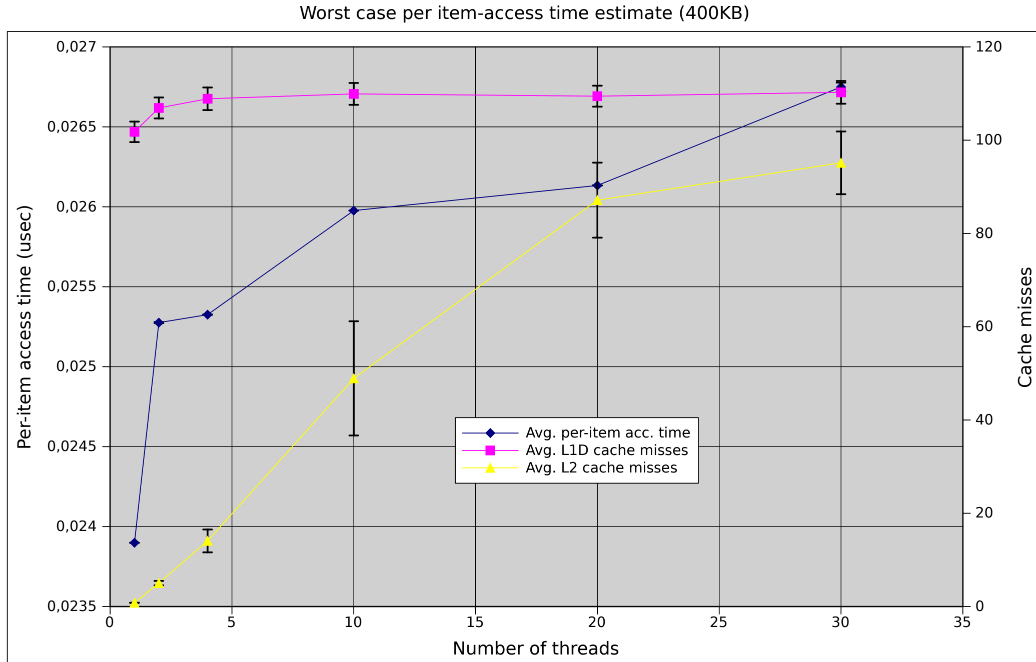


Figure 3.6: Worst case per-item access time estimate (400KB).

As we can see, the experiment starts with a situation like the one at the end of the previous test: one task with a working set of 400KB (L1D cache misses are obviously at the max rate). As the number of tasks increases we start to see an increasing number of L2 cache misses as well. At the same time the per-item access time goes beyond $0.026\mu s$, since L2 cache misses are heavier than the L1D one for the system.

3.4.3 Experiments on Random Tasks Sets

With the per-item access time estimate in our hands, we are now able to conduct several experiments in order to prove the goodness of our implementation. The focus of the following series of tests will be to show that

introducing tasks migrations doesn't afflict the system behaviour with a too heavy cache-related overhead.

Our interest is to compare the G-EDF behaviour with the P-EDF and the G-RM ones, where with G-RM we refer to the possibility to implement the Rate Monotonic scheduling algorithm [28] in a global way simply utilizing the `SCHED_FIFO` scheduling class (present in the stock Linux kernel). Rate Monotonic is in fact a static priority real-time algorithm, and to implement it is sufficient to assign at each tasks a priority on the basis of its period: the shorter the period, the higher is the task's priority. Then, if we not set tasks' affinities, they will be able to migrate between the CPUs of the system.

We divided the set of experiments in two distinct parts. In the first we compare G-EDF and P-EDF, in the second G-EDF and G-RM. At this purpose two groups of randomly generated tasks sets has been built; tasks sets from the first group passed are G-EDF schedulable and a first-fit allocation among the CPUs has been found for the tasks; tasks sets from the second group, instead, passed G-EDF and G-RM schedulability tests. Inside each group another three subgroups has been created in order to analyse system's behaviour with an high, medium or low load. Periods has been generated with an uniform distribution between $1ms$ and $100ms$. The number of tasks in a tasks set has been varied as well, this way we have tasks sets composed by 2, 4, 8 and 16 tasks (an higher number of soft real-time tasks beeing not interesting on a dual-core CPU).

In order to better summarize, we have:

- 50 tasks sets with a fixed number of tasks, for a total of 200 tasks sets for each subgroup and 600 for each group;
- each of these tasks sets is scheduled two times, e.d. the first time with G-EDF and the second with P-EDF;
- each experiment runs for 10 seconds, this not affects the credibility of the test since in the worst case (a task with period equal to $100ms$) we have 100 jobs per task;

It is to be said that in more than 6 hours of experiments we haven't

experienced crashes, so we can certainly state that our patches to the stock Linux kernel doesn't affect system's stability.

In order to analyze the collected data we will refer to three metrics of interest:

- Slack Time (Tardiness) / Period: if a job completes its work before the end of the period, the interval of time from the end of the execution time and the end of the period is called *slack time*, if, instead, a task's job goes beyond the end of its period (missing its deadline) that amount of time is called *tardiness*. To not experiment tardiness, on a schedulable task set, will be a sign that the scheduling cache-related overhead is negligible. Slack time has been related to periods as to normalize an otherwise too variable value.

- Number of L1D cache misses / Execution Time: since a global scheduling algorithm may introduce additional cache misses compared to a partitioned one (or to a different algorithm), it is interesting to look at how many L1D cache misses occurs varying the number of tasks concurrently executing. That value is normalized by the execution time of each task, as a longer job normally generates more misses.

- Percentage error between expected and real execution time: even if this will not be a precise metrics (timers' resolution may be a problem), beeing able to state that the percentage error is finite will be another prove of the goodness of our analysis.

G-EDF versus P-EDF

With the following analysis we compare, on the same test-bed, the same algorithm on its global and partitioned implementation. In practice we compare what we found already implemented as a patch of Linux and what we have added to it.

On the left of Figure 3.7 on the following page we find the G-EDF behaviour for high, medium and low system's utilizations. On the right it is shown the same for the P-EDF case. The two graphs are quite identical,

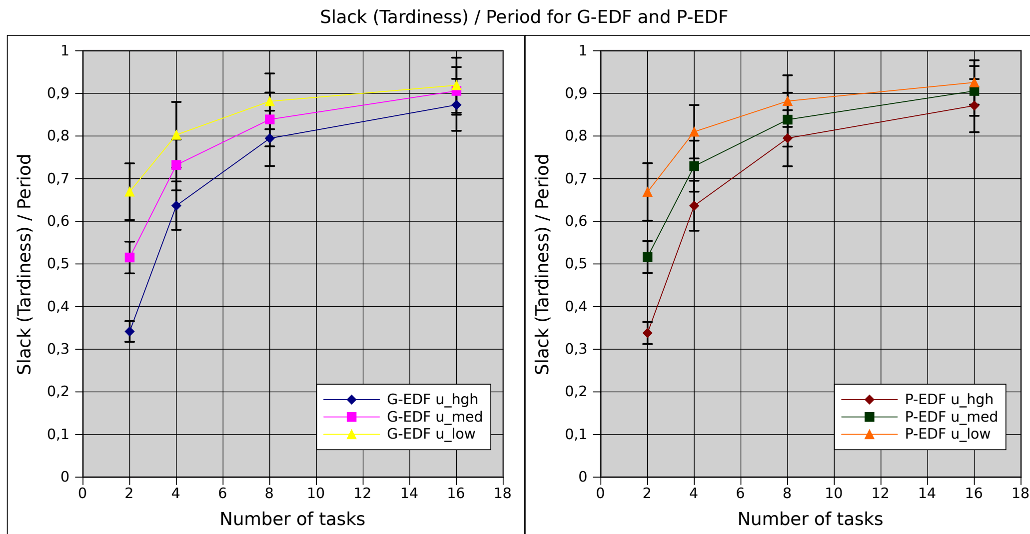


Figure 3.7: Slack time / Period for G-EDF and P-EDF.

we can't find appreciable differences between the global and the partitioned approach. The trend is an increase in slack/period as the number of tasks increases. This is due to the fact that, on equal terms of periods and total utilization, when less tasks have to use the same amount of system's bandwidth, they have to execute more than when there are more tasks. This involves that there will be less slack time and so the ratio will be small. When the number of task increases each task executes for less time and has a larger slack time, the slack/period ratio is bigger. Since we do not see tardiness in the G-EDF case and that the trend is the same as in the P-EDF case, we can state that our implementation behaves well we running on real hardware.

A further confirmation comes from Figure 3.8 on the next page. From this graph we can see that the trend is the same for G-EDF and P-EDF as well. Tasks migrations do not introduce in the system to much more cache miss events than when there are only preemptions. We can also state, not surprisingly, that a bigger number of tasks, that concurrently operate on the cache, generates an higher number of cache miss. At a fixed number of tasks lower utilization curves are above the higher ones because tasks' execution time is lower so the ratio beeing bigger.

With Figure 3.9 on page 70 we conclude the G-EDF versus P-EDF case.

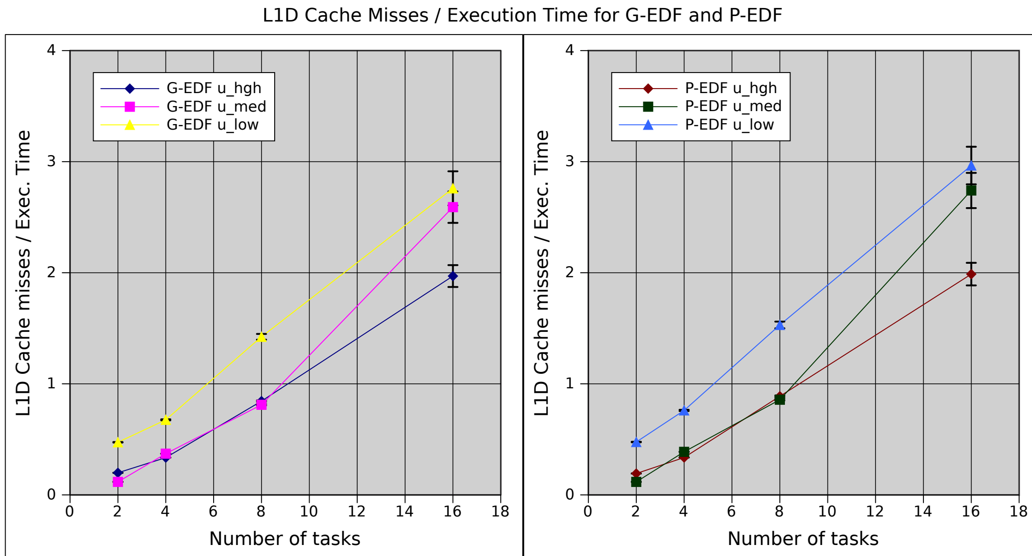


Figure 3.8: L1D cache misses / execution time for G-EDF and P-EDF.

As we stated before this kind of measures are not so reliable because high resolution timers have a bad work when dealing with very small interval of time (further causing a very big standard deviaton). The statement is confirmed if we look at the graphs. High utilization tasks sets all have a positive percentage error; since tasks that comes from that sets have all big execution times, the timers that manage the execution of those tasks behave well. When the execution time of tasks starts to be lower and lower, timers begin to behave bad and we see negative percentage errors. Anyway percentage error appears to be bounded between a +10% and a -15%, acceptable values for soft real time tasks.

G-EDF versus G-RM

It is now time to make a comparison between our implementation of the dynamic priority Earliest Deadline First scheduling algorithm and the static priority one Rate Monotonic. We present the results in the same order of the previous case, as to not confuse the reader.

In Figure 3.10 on the next page we find the Slack Time / Period trends for the G-EDF and G-RM cases. The two graphs are quite indentical, this

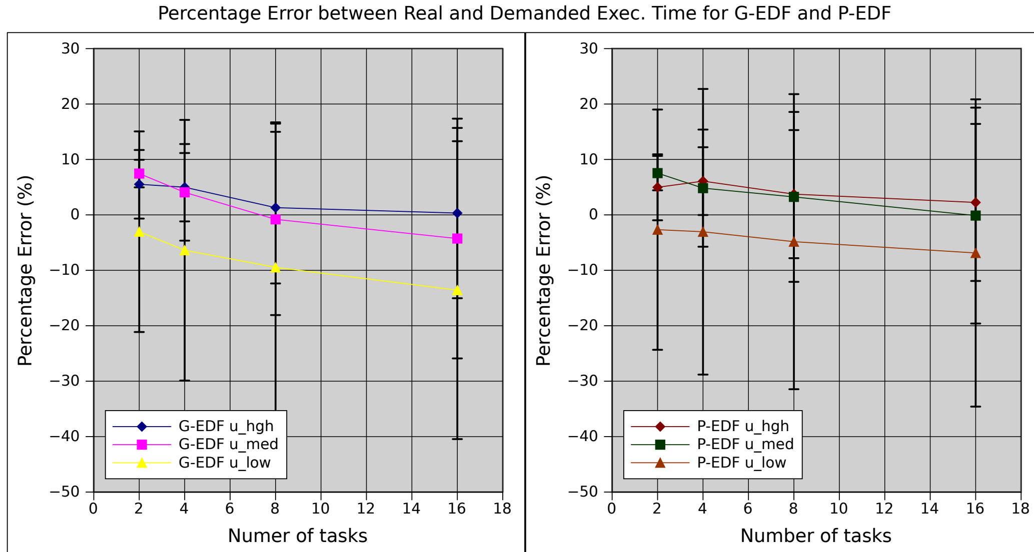


Figure 3.9: Percentage error between real and expected execution time for G-EDF and P-EDF.

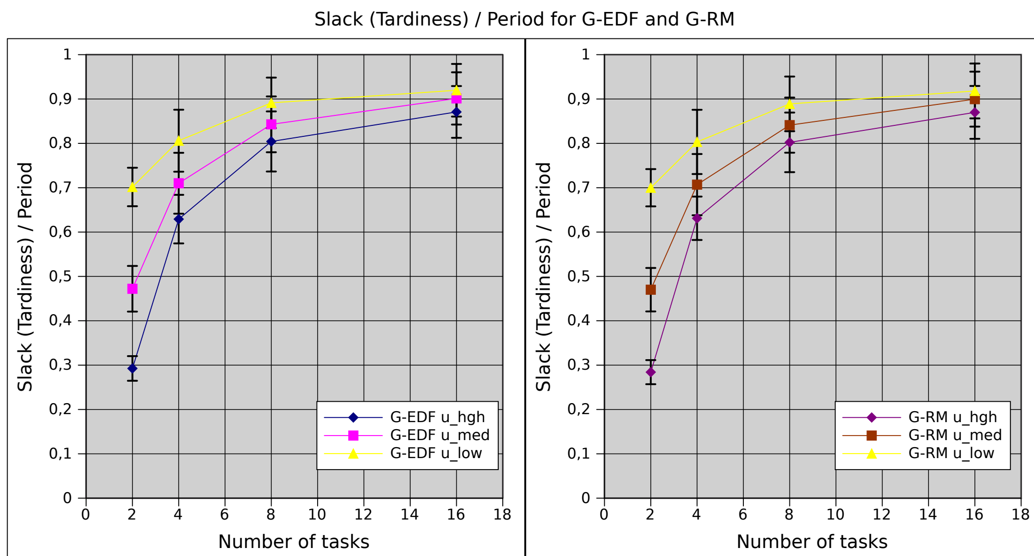


Figure 3.10: Slack time / Period for G-EDF and G-RM.

signifies that no one of the two algorithms is better considering cache-related scheduling overhead. In both cases the system doesn't experience tardiness proving that both the approaches are useful on scheduling soft real-time tasks.

The trends continues to be very similar for L1D cache misses as well. Figure 3.11 shows that fact, the predominant factor that drives the increase on the number of cache misses is the number of real-time tasks on the system.

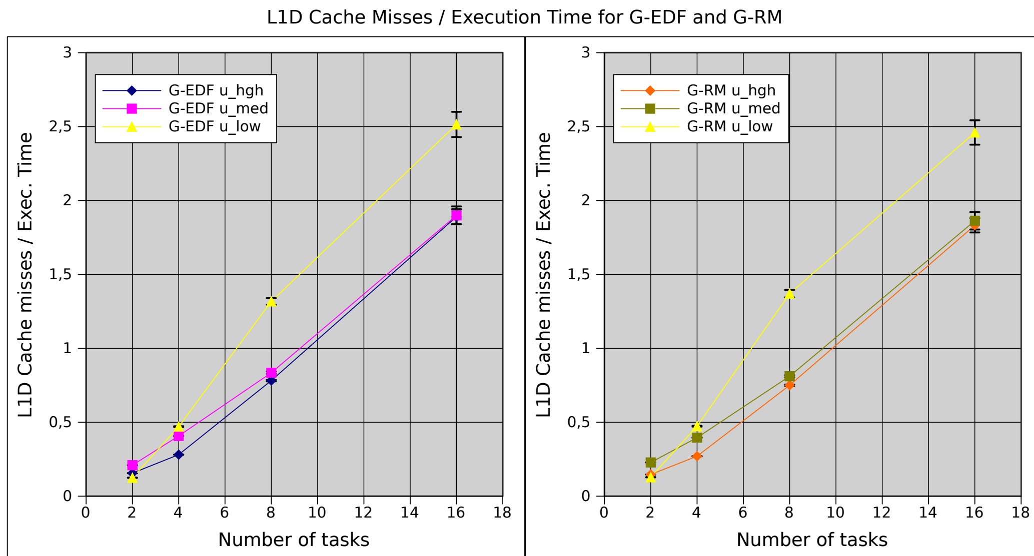


Figure 3.11: L1D cache misses / Execution time for G-EDF and G-RM.

The last graph (shown in Figure 3.12 on the next page) confirms that errors on estimate the execution time of a tasks are not related to a particular algorithm, but only to the not so fine resolution of systems' timers.

Concluding this comparison is important to remark that the use of a global approach on scheduling EDF tasks not influences the correct behaviour of the system. Moreover, since EDF is a dynamic priority scheduling algorithm, tasks can dynamically enter and leave the system and no a-priori knowledge of the system's evolution is needed. With our work we give to the user the opportunity to use a dynamical soft real-time system with all the positives that an open source kernel as Linux incorporates.

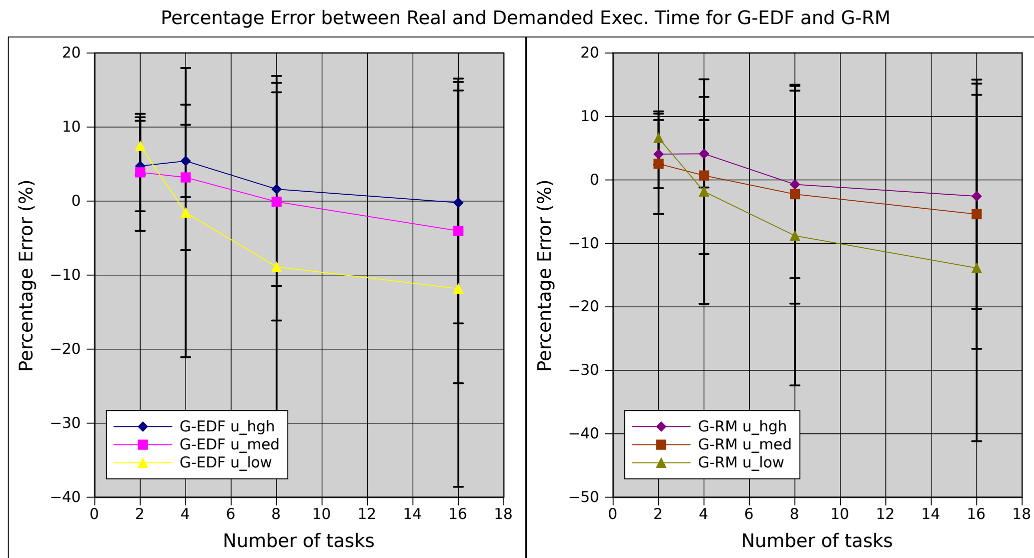


Figure 3.12: Percentage error between Real and Demanded Execution time for G-EDF and G-RM.

Chapter 4

Conclusions and Future Works

In this thesis we extended the SCHED_DEADLINE scheduling policy, an implementation of the Earliest Deadline First real-time scheduling algorithm in the Linux kernel. Linux has not been designed to be a Real-Time Operating System, but it is certain, at the time of writing, an increasing interest, from both academic and commercial worlds, on extending Linux to reach a sophisticated real-time support. Our focus was on SMP systems and our goal was to allow SCHED_DEADLINE tasks' migrations among the system's CPUs. For this purpose, we studied how the Linux scheduler works and how it is possible to implement new features inside its modular framework.
Several ways exist to implement migration of tasks, inside the Linux kernel we find an active and a passive approach. Real-time tasks need to execute as soon as they can, so we considered the passive approach not suitable to our needs and we have chosen the active one. Our implementation actively pushes and pulls tasks among the CPUs when needed, the desired behaviour should be to have the m earliest deadline tasks always running on the m CPUs of the system.

To prove the reliabilty of a working kernel with G-EDF tasks running on it and in order to analyze probable cache-related overheads (when a task migrates it lacks the so called cache affinity), we conducted several hours of experiments. First we estimated, with some micro-benchmarks, the amount of cache-related overhead a task may suffer in a worst case situation where

a lot of tasks works concurrently on the same processor. This has been done reading high resolution system's clocks and with the use of a library (PAPI) that allows to inspect performance counters of a CPU. With all these measures at hand we started to schedule periodic tasks. Two groups of random tasks sets were created by the use of a random tasks sets generator (we extended to our needs an already existing application). With one group we compared the G-EDF and P-EDF approaches, with the other the G-EDF and G-RM scheduling algorithms.

The results of the experimental analysis confirms the goodness of our implementation. Allowing tasks migration not introduces an amount of overhead that the system is practically unusable. With our work we have instead given to the Linux users an useful tool to develop dynamical soft real-time applications.

The research on this topic is all but over. Many more experimental analysis are possible and a further optimization of the migration mechanism may be interesting. It is remained outside from this thesis to investigate how, and if, the peculiarities of each task on a tasks set may influence the number of migrations. Tasks set can be distinguished by some quantities, such as, total utilization, number of tasks, maximum per-task bandwidth, ratio between light and heavy tasks, etc.... Therefore, it is interesting to see if the total number of migrations the system experiences can be related to one or more of the aforementioned quantities. A comparison between the number of tasks migrations generated by the Rate Monotonic scheduling algorithm and the Earliest Deadline First will be of great interest. With this kind of analysis it should be possible to understand if a particular scheduling approach can affect how much tasks migrate and if such an approach is influenced by a particular tasks set parameter.

Experiments should be conducted on different types of hardware platforms, such as, multicore with more than two cores, multiprocessors and NUMA systems. Real applications, as in [23] , should be tested on that kind of hardware in order to see how the system behaves on the real world.

An improvement on efficiency of the present implementation of the migration mechanism is possible as well. Several data structures should be

analyzed in order to minimize the amount of time spent on finding a suitable runqueue when a task has to migrate. Obviously, a practical comparison between those data structures will be possible only on large multiprocessors systems.

Using kernel tracers it should be possible to see if the present implementation strictly complies to the G-EDF algorithm. If it will not be the case it will be important to measure the amount of time the system is under a transient state and the reasons why of this unwanted situation.

Finally, it will be of a great interest to analyze how the G-EDF scheduling mechanism can work inside soft real-time systems based on *reservations* and resource sharing as FRESCOR or AQuoSA.

Appendix A

Source Code

A.1 SCHED_DEADLINE

The complete source code of the `SCHED_DEADLINE` scheduling policy is available as a git repository at this address:
`git://gitorious.org/sched_deadline/linux-deadline.git`
The home page of the project is hosted by Evidence at this address:
`http://www.evidence.eu.com/content/view/313/417/`

A.2 Micro-benchmarks

The source code of the micro-benchmarks and test programs developed in this thesis can be found as a git repository at this address:
`git://gitorious.org/sched_deadline/tests.git`

Ringraziamenti

Tante persone da ringraziare e poco tempo per farlo, spero di non dimenticare nessuno.

Desidero innanzitutto ringraziare i miei relatori, Prof. Giuseppe Lipari e Prof. Giuseppe Anastasi per l'aiuto e l'incoraggiamento continuo durante tutto il lavoro svolto per questa tesi. Un ringraziamento particolare va poi a Dario Faggioli, mio tutor per tutti questi mesi. Quello che ha fatto per me (i consigli su come iniziare a modificare il suo codice e le nottate in bianco passate a lavorare sodo per riuscire a finire la tesi in tempo per questo appello di laurea) è stato davvero indispensabile, spero di poter ricambiare presto. Ringrazio inoltre tutti i ragazzi del ReTiS Lab per i consigli, la compagnia e la selezione musicale.

Un saluto ed un abbraccio a tutti gli amici che mi hanno accompagnato in questi anni passati all'Università. Quelli con cui ho iniziato l'avventura e che non mi hanno mai abbandonato, il Cica, Ale, Prasao, Federico e tutti gli altri con cui ho legato con il tempo, tra i quali, Matte, Azzarino e Giuliano. Non dimentico certo Luigi, mio coinquilino per due anni. Infine tutti i ragazzi con cui continuo a giocare a calcetto tutte le settimane e il fantastico gruppo di TangoMiAmor. Grazie a tutti, mi avete aiutato a distrarmi nei momenti faticosi e a divertirmi in quelli spensierati.

Probabilmente non ce ne sarebbe bisogno, loro lo sanno già, ma quello che mi hanno dato è troppo grande per non ricordarlo. Sto parlando della mia famiglia. Senza di loro non sarei niente e tutto questo non sarebbe stato nemmeno lontanamente possibile. Con le lacrime agli occhi come uno scemo mentre scrivo queste ultime righe in laboratorio vi voglio dire grazie di cuore, nonno Gualtiero, mamma Graziella, babbo Ivano e Vladi, vi voglio bene.

Grazie anche a Maria, Franca, Paolo, Roberta, Martina, Daniele, Zia, Zio, Sara e Massimiliano perché avete fatto sempre il tifo per me.

E poi ci sei tu Claudia, amoramiaunicaspecialetanguerapimpadolcemia, sei tutta la mia vita.

Grazie a tutti!

Bibliography

[1] L. Abeni and G. Buttazzo. Resource reservations in dynamic real-time systems. *Real-Time Systems*, 27(2):123–165, 2004.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[3] L. Abeni and G. Lipari. Implementing resource reservations in linux. In *Proc. of Fourth Real-Time Linux Workshop*, 2002.

[4] AQuoSA. Aquosa - "adaptive quality of service architecture" (for the linux kernel). `http://aquosa.sourceforge.net/index.php`.

[5] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 214–224, December 2009.

[6] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169, December 2008.

[7] G. Buttazzo. *Sistemi in Tempo Reale*. Pitagora Editrice Bologna, 2006.

[8] J. Calandrino, J. Anderson, and J. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-time Systems*, pages 247–256, July 2007.

[9] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, chapter 30: A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms. Chapman Hall/CRC Press, 2004.

[10] Johnathan Corbet. Cfs group scheduling. `http://lwn.net/Articles/240474/`.

[11] Johnathan Corbet. Deadline scheduling for linux. `http://lwn.net/Articles/356576/`.

[12] Johnathan Corbet. Scheduling domains. `http://lwn.net/Articles/80911/`.

[13] Mathieu Desnoyers. Using the linux kernel tracepoints. Documentation/trace/tracepoints.txt.

[14] U Devi and J. Anderson. Tardiness bounds for global edf scheduling on multiprocessor. In *Proceedings of the 26th IEEE Real-time Systems Symposium*, pages 330–341, 2005.

[15] Linux documentation. Design of the cfs scheduler. Documentation/scheduler/sched-design-CFS.txt.

[16] L. Dozio and P. Mantegazza. Real-time distributed control using rtai. In *Sixth IEEE Internation Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '03)*, Hakodate, Hokkaido, Japan, May 2003.

[17] U. Drepper. What every programmer should know about memory. 2007.

[18] Community Research European Commission. Sixth framework programme. `http://ec.europa.eu/research/fp6/index_en.cfm`.

[19] Dario Faggioli. sched: Sched_deadline v2. `http://lkml.org/lkml/2010/2/28/107`.

[20] Dario Faggioli, Michael Trimarchi, Fabio Checconi, and Scordino Claudio. An edf scheduling class for the linux kernel. In *Proceedings of the 11th Real-Time Workshop (RTLW)*, October 2009.

[21] FRESCOR. Framework for real-time embedded systems based on contracts. `http://www.frescor.org/index.php?page=FRESCOR-homepage`.

[22] P. Gerum. The xenomai project, implementing a rtos emulation framework on gnu/linux. Nov. 2002.

[23] C. Gough, S. Suresh, and K. Chen. Kernel scalability - expanding the horizon beyond fine grain locks. *Linux Symposium*, One, 2007.

[24] PREEMPT_RT group. Config_preempt_rt patch set. `http://www.kernel.org/pub/linux/kernel/projects/rt/`.

[25] Wind River Linux. `http://www.windriver.com/products/linux/`.

[26] G. Lipari and C. Scordino. Linux and real-time: Current approaches and future oppostunities. *IEEE International Congress ANIPLA*, 2006.

[27] LITMUS^{RT}. Linux testbed for multiprocessor scheduling in real-time systems. `http://www.cs.unc.edu/~anderson/litmus-rt/`.

[28] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[29] Ingo Molnar. Modular scheduler core and completely fair scheduler [cfs]. `http://lkml.org/lkml/2007/4/13/180`.

[30] montavista. `http://www.mvista.com/real_time_linux.php`.

[31] OProfile. A system profiler for linux. `http://oprofile.sourceforge.net/news/`.

[32] PAPI. Performance application programming interface. http://icl.cs.utk.edu/papi/index.html.

[33] OCERA Project. Open components for embedded real-time applications. http://www.ocera.org/index.html.

[34] I. Ripoll, P. Pisa, L. Abeni, P. Gai, A. Lanusse, S. Sergio, and B. Privat. Wp1 - rtos state of the art analysis: Deliverable d1.1 - rtos analysis. OCERA, 2006.

[35] Steven Rostedt. New rt balancing version 4. http://lkml.org/lkml/2007/11/20/558.

[36] RTAI. Rtai - the realtime application interface for linux from diapm. https://www.rtai.org/.

[37] C. Scordino and G. Lipari. A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks. *IEEE Transaction on Computers*, 2006.

[38] Evidence S.r.l. Sched_deadline. http://www.evidence.eu.com/content/view/313/390/.

[39] John A. Stankovic, K. Ramamritham, M. Spuri, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[40] techradar.com. Arm: dual core mobiles coming in 2010. http://www.techradar.com/news/phone-and-communications/mobile-phones/arm-dual-core-mobiles-coming-in-2010-645417?src=rss&attr=all.

[41] Timesys. https://linuxlink.timesys.com/3/Linux.

[42] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors. In *Proceedings of the 26th IEEE Real-time Systems Symposium*, pages 311–320, 2005.

[43] V. Yodaiken. The rtlinux manifesto. In *Proceeding of the Fifth Linux Expo*, Raleigh, North Carolina, Mar. 1999.