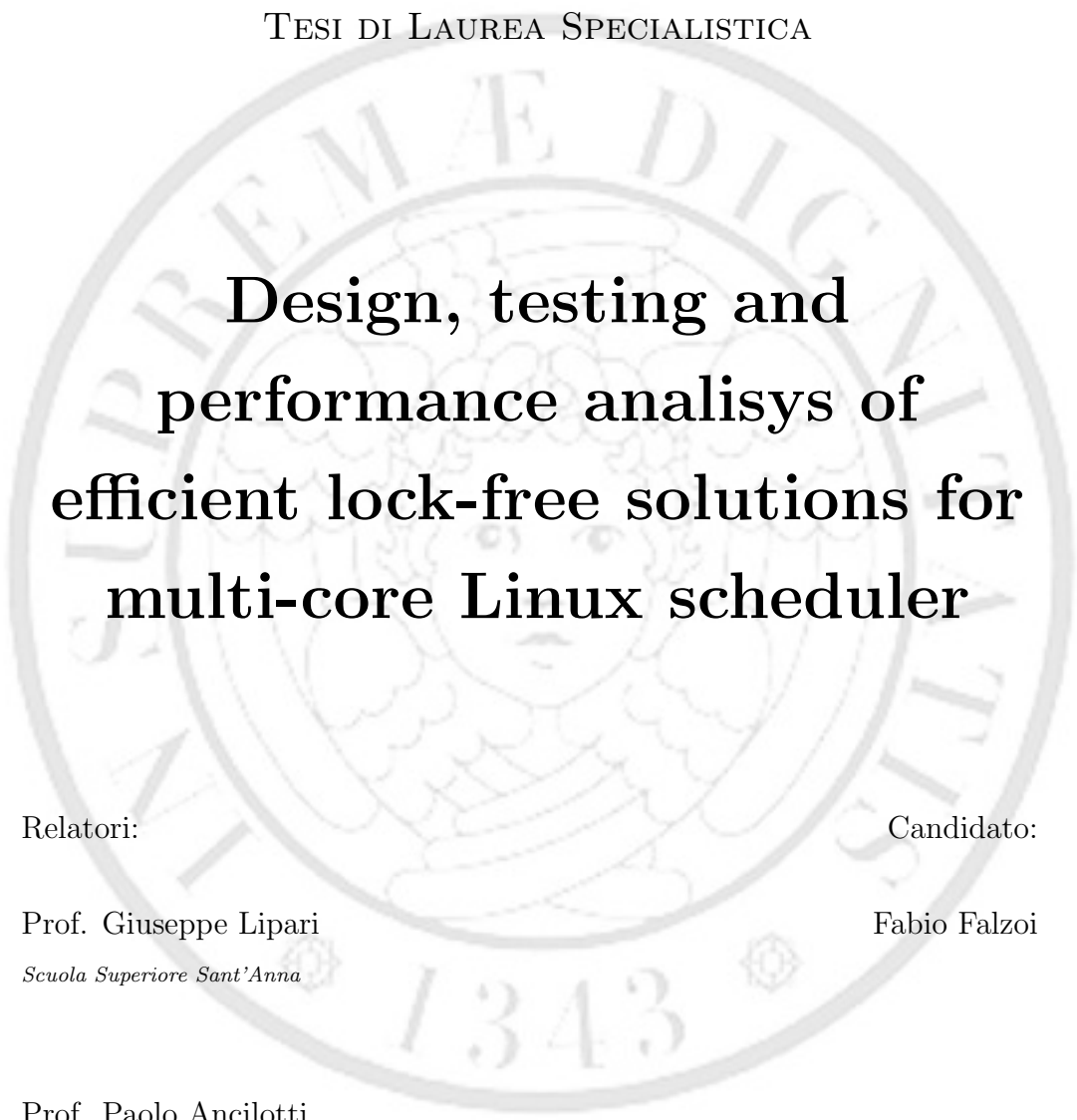


UNIVERSITÀ DI PISA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

TESI DI LAUREA SPECIALISTICA



**Design, testing and
performance analysis of
efficient lock-free solutions for
multi-core Linux scheduler**

Relatori:

Prof. Giuseppe Lipari

Scuola Superiore Sant'Anna

Prof. Paolo Ancilotti

Scuola Superiore Sant'Anna

Candidato:

Fabio Falzoi

ANNO ACCADEMICO 2011/2012

*In loving memory of my mother
and my grandfather.*

Abstract

Multiprocessor systems are now the de facto preferred computing platform for many application domains, including personal computers and server workstations.

The benefits of multi-core technology in terms of increased computational power with a reduced energy consumption, are now being used for more implementing efficient embedded devices and personal appliances like smart phones and tablets.

A popular OS like Linux, which was not originally designed to be a Real-Time Operating System (RTOS), is now being used for embedded real-time systems with multi-core platforms. Consequently, many flavors of Linux now include a real-time scheduler. One recent example of real-time scheduler for Linux is the SCHED_DEADLINE patch, an implementation of the popular “Earliest Deadline First” algorithm. Such scheduler can be useful also for large many-core server workstations, because it helps to control the quality of service of the user requests. For this reason, it is important that the scheduler implementation to be efficient and scale well with the number of processors.

In this thesis, I present two original contributions to the area of real-time scheduling in the Linux kernel. First, I present PRACTISE, a tool to develop, debug, test and analyse real-time scheduling data structures in user space. Unlike other similar tools, PRACTISE executes code in parallel, allowing to test and analyse the performance of the code in a realistic multiprocessor scenario.

Then, I present several data structures for implementing a distributed queue to efficiently support global scheduling in a large multi-core: max-heap and min-heap, skip-list with a flat-combiner strategy, and a novel algorithm called *fast-cache*. I compare the different data structures and algorithms using both PRACTISE and directly in the kernel.

Contents

Introduction	x
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
1.2 The Linux real-time scheduler	4
1.2.1 SCHED_FIFO and SCHED_RR	5
1.2.2 Multiprocessor support	6
1.2.3 Linux scheduler multiprocessor support in real-time scheduling class	8
1.2.4 Real-time load balancing algorithm	8
1.2.5 Real-time scheduler data structures and concepts	9
1.2.6 Root domains	11
1.2.7 CPU priority management	12
1.2.8 Details of the <i>Push</i> scheduling algorithm	13
1.2.9 Details of the <i>Pull</i> scheduling algorithm	14
1.3 State of the art of Real-Time scheduling on Linux	15
1.3.1 RTLinux, RTAI and Xenomai	15
1.3.2 PREEMPT_RT	17
1.3.3 OCERA	17
1.3.4 AQuoSA	18
1.3.5 FRESCOR	18
1.3.6 LITMUS ^{RT}	19

1.4	EDF and CBS theory	19
1.4.1	Earliest Deadline First	20
1.4.2	Constant Bandwidth Server	22
1.4.3	EDF scheduling on SMP systems	23
1.5	The SCHED_DEADLINE scheduling class	24
1.5.1	Main Features	25
1.5.2	Interaction with Existing Policies	26
1.5.3	Current Multiprocessor Scheduling Support	26
1.5.4	SCHED_DEADLINE Push implementation	29
1.5.5	Max-heap cpudl data structure for push operation	33
1.5.6	SCHED_DEADLINE Pull implementation	35
1.5.7	Task Scheduling	37
1.5.8	Usage and Tasks API	38
2	Synchronization mechanisms analysis	40
2.1	Kernel locking techniques	40
2.1.1	SMP and UP Kernel	41
2.1.2	Atomic operators	41
2.1.3	Spinlocks	42
2.1.4	Semaphores	43
2.1.5	Reader/Writer locks	44
2.2	Memory barriers	45
2.2.1	Abstract memory access model	46
2.2.2	CPU guarantees	48
2.2.3	Behaviour and varieties of memory barriers	49
2.2.4	SMP barriers pairing	51
2.2.5	Explicit Linux kernel barriers	52
2.2.6	Implicit kernel memory barriers	54
2.3	Flat combining	54
3	New solutions for task migration	56
3.1	Skip list	56
3.1.1	Skip List structure and asymptotic complexity	57

<i>CONTENTS</i>	vi
3.1.2 cpudl skip list implementation	58
3.2 Lock-free skip list	60
3.3 Bitmap flat combining	61
3.3.1 Flat combining implementation details	61
3.3.2 cpudl bitmap flat combining implementation	62
3.4 Fastcache	64
3.5 Improved pull algorithm	67
4 PRACTISE framework	70
4.1 Tools for Linux kernel development	70
4.1.1 LinSched	71
4.1.2 LITMUS ^{RT}	72
4.1.3 KVM + GDB	72
4.2 PRACTISE architecture	72
4.2.1 Ready queues	73
4.2.2 Locking and synchronization	74
4.2.3 Event generation and processing	74
4.2.4 Data structures in PRACTISE	77
4.3 Performance analysis with PRACTISE	78
4.4 Evaluation	81
4.4.1 Porting to Linux	81
5 Experimental Results	84
5.1 Experiments with PRACTISE	85
5.2 Kernel Experiments	90
5.3 Comparison between max-heap and skip list	90
5.4 Improved Pull algorithm performance	93
5.5 Bitmap flat combining performance	95
5.6 Fastcache performance	100
6 Conclusions and Future Work	106
A Code listings	108
A.1 cpudl skip list implementation	108

<i>CONTENTS</i>	vii
A.2 cpudl bitmap flat combining implementation	114
A.3 cpudl fastcache implementation	129
A.4 Improved pull algorithm	135
Acknowledgments	137

List of Figures

1.1	The Linux modular scheduling framework.	3
1.2	The CFS runqueue.	5
1.3	An EDF scheduling example.	21
1.4	The Linux modular scheduling framework with SCHED_DEADLINE.	27
1.5	<i>cpudl</i> structure for <i>push</i> operation.	35
2.1	An abstract model of a multiprocessor system.	46
2.2	A sequence of memory operations where SMP barrier pairing is required.	52
3.1	An example skip list.	57
3.2	<i>cpudl</i> structure for <i>pull</i> operation.	68
4.1	Comparison using <i>diff</i>	82
5.1	Global data structure modify	86
5.2	Global data structure query	87
5.3	Global data structure <i>cpupri</i> modify	88
5.4	Global data structure <i>cpupri</i> query	88
5.5	Global data structure <i>cpudl</i> modify	89
5.6	Global data structure <i>cpudl</i> query	89
5.7	<i>set</i> operation on max-heap and skip list kernel	91
5.8	<i>find</i> operation on max-heap and skip list kernel	92
5.9	<i>set</i> operations number	93
5.10	<i>find</i> operations number	94
5.11	Number of task migrations due to push operation	95

5.12	Number of task migrations due to pull operation	96
5.13	Bitmap flat combining push performance	97
5.14	Bitmap flat combining pull performance	98
5.15	Number of successfull push operations	99
5.16	Fastcache push performance	101
5.17	Fastcache pull performance	102
5.18	Number of task migrations due to push operation	103
5.19	Number of task migrations due to pull operation	104

Introduction

Multiprocessor systems are nowadays de facto standard for both personal computers and server workstations. Benefits of dual-core and quad-core technology is also common in embedded devices and cellular phones as well. In fact, raw increases in computational power is no more the answer for overall better performance: the energy efficiency is a primary concern, that can't be ignored at any level of a system design, from hardware to software. Regarding the hardware layer, multicore and multiprocessors technologies surely gave an answer to that issue, but without a proper software design, the scalability of the entire system may suffer.

The role of the operating system scheduler is fundamental while managing the threads of execution: a sub-optimal schedule may lead to high latency and very poor overall performance. If real time tasks, characterized by strictly timing constraints, are also considered, we can easily understand that finding an optimal schedule is far from trivial.

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 its own issues, the common trend seems to be a considerable interest in using Linux for real-time and control applications.

But 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.

In a recent paper [14], Dario Faggioli and others proposed an implementation of the “Earliest Deadline First” (EDF) algorithm in the Linux kernel. In order to extend stock Linux kernel’s features a new scheduling policy has been created: `SCHED_DEADLINE`. Later, Juri Lelli extended that scheduling policy to add processes migration between CPUs [19]. This allowed to reach full utilization of the system in multicore and multiprocessor environment. While the proposed implementation is indeed effective, a problem of scalability arises when the scheduler has to dynamically assigns real-time tasks to an high number of online CPUs. All the scheduler shared data structures are potential performance bottlenecks: the contention to manipulate the data structure may increase a lot, leading to unpredictable and unbounded latencies.

Unfortunately, the development of new solutions to manage concurrency in kernel space is far from trivial: when the number of parallel scheduler instances increases the common tools used for debugging are not so effective.

In this thesis, we propose PRACTISE, a tool for performance analysis and testing of real-time multicore schedulers for the Linux kernel. PRACTISE enables fast prototyping of real-time multicore scheduling mechanisms, allowing easy debugging and testing of such mechanisms in user-space. Thanks to PRACTISE we developed a set of innovative solutions to improve the scalability of the processes migration mechanism. We will show that, with those modifications, not only a better scalability has been reached, but also a schedule closer to *G-EDF* policy of the tasks has been achieved.

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 improve the `SCHED_DEADLINE` implementation, 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. Finally, we will discuss in great detail about the current implementation of the task migrations algorithm in `SCHED_DEADLINE` scheduling class.

Chapter 2 (**Synchronization Mechanisms Analysis**) gives a detailed explanation of the available mechanisms to manage concurrent accesses on a shared data structure. In particular, we will refer to the synchronization techniques in Linux kernel. Finally, we will explain a recently proposed framework that aims to improve performance for shared data structures accessed in parallel by a significant number of threads.

In Chapter 3 (**New Solutions for Task Migration**) we present a set of new solutions for the task migration algorithms. We will show the main idea behind each of those to explain why such a design was chosen.

Chapter 4 (**PRACTISE Framework**) contains the details of PRACTISE implementation. Here we will explain how PRACTISE was designed and how it can be used to facilitate the development of new kernel code. In the last part of the chapter we focus on the ability of PRACTISE to predict the relative performance of the various algorithm simulated with it.

Chapter 5 (**Experimental Results**) contains the graphs that show the results of our experiments conducted with the Linux kernel. We present the results of each new algorithm discussed above, explaining why a certain performance trend is achieved. Doing so, we will point out the main advantages and also the disadvantages of each solution.

Finally, in Chapter 6 (**Conclusions and Future Works**), we sum up results and suggest possible future extensions to the code as well as 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, and the process scheduler (or simply the *scheduler*) is a subsystem of the kernel that assigns processor time to the runnable processes.

In a single processor machine, the scheduler gives 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 affects 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.

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

1.1.1 Modular scheduling framework

The current version of the Linux scheduler has been designed and implemented by Ingo Molnar [24] as a modular framework that can easily be extended. Each scheduler module is a *scheduling class* that encapsulate specific scheduling policies details.

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 scheduler hooks is:

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

Three “*fair*” scheduling policies (`SCHED_NORMAL`, `SCHED_BATCH`, `SCHED_IDLE`) and two *real-time* scheduling policies (`SCHED_RR`, `SCHED_FIFO`) are currently implemented in the Linux scheduler. The situation is depicted in Figure 1.1 on the following page.

² Defined in `include/linux/sched.h`.

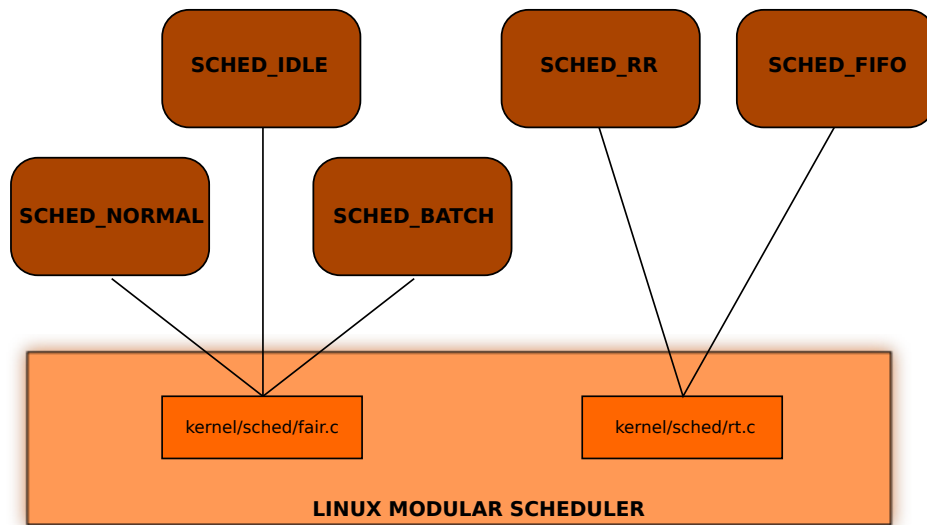


Figure 1.1: The Linux modular scheduling framework.

1.1.2 Scheduling entities, tasks and runqueues

All data used by the scheduler to implement any scheduling policy 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 CFS⁴ scheduler uses to carry out his job. The concept of *scheduling entity* is essentially “something to be scheduled”, which might not be a process (e.g. tasks groups [7]).

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

- `volatile long state`: it describes the task’s state. It can assume three values (`-1`, `0`, `>0`) depending on the task respectively being *unrunnable*, *runnable* or *stopped*.
- `const struct sched_class *sched_class`: it binds the task to his scheduling class.
- `struct sched_entity se`, `struct sched_rt_entity rt`: it

³Defined in `include/linux/sched.h`.

⁴*Completely Fair Scheduler*, the default Linux scheduler, see [10].

⁵Defined in `include/linux/sched.h`.

contains *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, we have 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.

To better understand the inner working of the scheduler, 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 left-most 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 practice: 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 presents the structure of the run-queues.

1.2 The Linux real-time scheduler

Linux has been designed to be a general-purpose operating system (GPOS), therefore it presents some issues, like unpredictable latencies, limited support

⁶Defined in `kernel/sched.h`, with all runqueue related things.

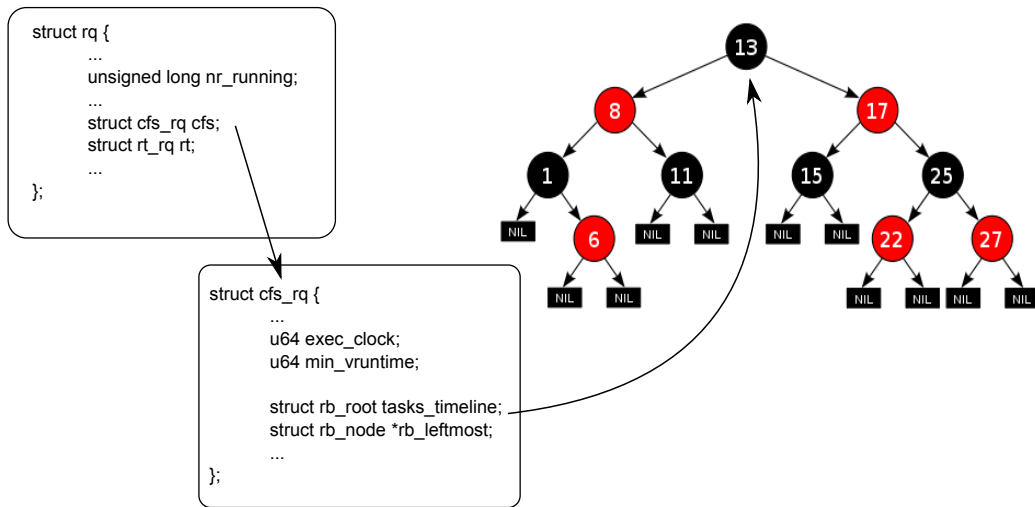


Figure 1.2: The CFS runqueue.

for real-time scheduling, and coarse-grain timing resolution that might be a problem for real-time application [20]. The main design goal of the Linux kernel has been (and still remains) to optimise the average throughput (i.e., the amount of “useful work” done by the system in the unit of time).

Since Linux is a POSIX-compliant operating system, the Linux scheduler must also provide `SCHED_FIFO` and `SCHED_RR` scheduling algorithms. These algorithms are actually implemented inside the `SCHED_RT` scheduling class, and so they represent the part of Linux kernel code dedicated to real-time tasks management. In this section we provide a brief explanation of those classes, with an inspection to the implementation code, with particular reference to multiprocessor systems support.

1.2.1 `SCHED_FIFO` and `SCHED_RR`

`SCHED_FIFO` and `SCHED_RR` are simple fixed-priority policies. According to the POSIX standard⁷, `SCHED_FIFO` is a strictly first in-first out (FIFO) scheduling policy. This policy contains a range of at least 32 priorities (actually, 100 inside Linux). Tasks scheduled under this policy are chosen from a thread list ordered according to the time its tasks have been in the list

⁷IEEE Std 1003.1b-1993

without being executed. The head of the list is the task that has been in the list the longest time; the tail is the task that has been in the list the shortest time.

SCHED_RR is a round-robin scheduling policy with a per-system time slice, named *time quantum*. This policy contains a range of at least 32 priorities and is identical to the SCHED_FIFO policy with an additional rule: when the implementation detects that a running process has been executed for an interval equal or greater than the time quantum, the task becomes the tail of its task list, and the head of that task list is removed and made a running task.

Both SCHED_FIFO and SCHED_RR unfortunately diverges from what the real-time research community refer to as “realtime” [5]. 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.

1.2.2 Multiprocessor support

Since now, we have not addressed the issue of 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.

A multiprocessor Linux kernel (that is, one configured with CONFIG_SMP flag set, see Section 2.1.1 for more details) has additional fields into the aforementioned structures in comparison to a uniprocessor one.

In `struct sched_class` we find:

- `select_task_rq(...)`: it is 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(...)`: it checks the given CPU to ensure that it is balanced within scheduling domain (see below); if not, attempts to move tasks. This function is not implemented by every scheduling class.

- `pre_schedule(...)`: it is called inside the main `schedule` routine; performs the scheduling class related jobs to be done before the actual scheduling.
- `post_schedule(...)`: like the previous routine, but after the actual scheduling.
- `task_woken(...)`: it is called when a task wakes up, there could be things to do if we are not going to schedule soon.
- `set_cpus_allowed(...)`: it 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 objectives of a multiprocessor (non real-time) scheduler is to 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* [8] have been introduced into the Linux kernel.

A *scheduling domain* (`struct sched_domain`⁸) 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.

⁸Defined in `include/linux/sched.h`.

Furthermore, the *scheduling domains* architecture helps to reduce the contention for scheduler shared data structures, so to avoid significant lowering of performance in a very large multiprocessor system.

1.2.3 Linux scheduler multiprocessor support in real-time scheduling class

In a multi-core environment, where we have N available CPUs, only the N highest-priority tasks will be running at any given point in time. When a task is runnable, the scheduler must ensure that it be put on a runqueue best suited for it, that is, the real-time scheduler has to ensure system-wide strict real-time priority scheduling.

Unlike non-real-time systems where the scheduler needs to look only at its runqueue of tasks to make scheduling decisions (or, at most, it needs to run an inter-processor load balancing routine very infrequently), a real-time scheduler makes global scheduling decisions, taking into account all the tasks in the system at any given point. Furthermore, real-time tasks balancing also has to be performed frequently.

Task balancing can introduce cache thrashing and contention for global data and can degrade throughput and scalability. A real-time task scheduler would trade off throughput in favor of correctness, but at the same time, it must ensure minimal task migration.

1.2.4 Real-time load balancing algorithm

In this section we will detail the strategy used by Linux to balance real-time tasks across CPUs. This strategy has been introduced as a trade-off between global theoretical scheduling policy adherence and performance scalability.

The real-time scheduler adopts an active *push-pull* strategy developed by Steven Rostedt and Gregory Haskins for balancing tasks across CPUs. The scheduler has to address several scenarios:

1. Where to place a task optimally on wakeup.

2. What to do with a lower priority task when it wakes up but is on a runqueue running a task of higher priority.
3. What to do with a low priority task when a higher priority task on the same runqueue wakes up and preempts it.
4. What to do when a task lowers its priority and thereby causes a previously lower priority task to have the higher priority.

A pre-balance algorithm is used in case 1 above, often leading to a push operation. A push operation is also initiated in cases 2 and 3 above. The push algorithm considers all the runqueues within its scheduling domain (see 1.2.2) to find the one that is of a lower priority than the task being pushed.

A pull operation is performed for case 4 above. Whenever a runqueue is about to schedule a task that is lower in priority than the previous one, it checks to see whether it can pull tasks of higher priority from other runqueues. Real-time tasks are affected only by the push and pull operations. The CFS load-balancing algorithm does not handle real-time tasks at all, as it has been observed that the CFS load-balancing algorithm pulls real-time tasks away from runqueues to which they were correctly assigned, inducing unnecessary latencies.

1.2.5 Real-time scheduler data structures and concepts

As stated in Section 1.1.2, the main per-CPU runqueue data structure `struct rq`, holds a structure `struct rt_rq`, that encapsulates information about the real-time tasks placed on the per-CPU runqueue. In Listing 1.1 we can see the most relevant fields.

Listing 1.1: `struct rt_rq`

```
struct rt_rq {
    struct rt_prio_array active;
    ...
    unsigned int rt_nr_running;
    unsigned long rt_nr_migratory;
    unsigned long rt_nr_uninterruptible;
    struct {
```

```
    int curr;
    int next;
} highest_prio;
int overloaded;
};
```

Real-time tasks have a priority in the range of 0-99. These tasks are organized on a runqueue in a priority-indexed array `active`, of type `struct rt_prio_array`. An `rt_prio_array` consists of an array of subqueues. There is one subqueue per priority level. Each subqueue contains the runnable real-time tasks at the corresponding priority level. There is also a bitmask corresponding to the array that is used to determine effectively the highest priority task on the runqueue.

`rt_nr_running` and `rt_nr_uninterruptible` are counts of the number of runnable real-time tasks and the number of tasks in the `TASK_UNINTERRUPTIBLE` state, respectively.

`rt_nr_migratory` indicates the number of tasks on the runqueue that can be migrated to the other runqueues. Some real-time tasks are bound to a specific CPU, so, even if the runqueue is overloaded (that is, the runqueue holds more than one real-time task), that tasks cannot be pushed away or pulled from another CPUs. Unfortunately, the other CPUs cannot determine this without the overhead of locking several data structures. This can be avoided by maintaining a count of the number of tasks on the runqueue that can be migrated to other CPUs. When a task is added to a runqueue, the hamming weight of the `task->cpus_allowed` mask is looked at (cached in `task->rt.nr_cpus_allowed`). If the value is greater than one, the `rt_nr_migratory` field of the runqueue is incremented by one. The `overloaded` field is set when a runqueue contains more than one real-time task and at least one of them can be migrated to another runqueue. It is updated whenever a real-time task is enqueued on a runqueue. The `highest_prio` field is a structure caching the priority of the two highest priority tasks queued on the runqueue. Also this structure is updated whenever a task is enqueued on a runqueue.

1.2.6 Root domains

As mentioned before, because the real-time scheduler requires several system-wide resources for making scheduling decisions, scalability bottlenecks appear as the number of CPUs increase, due to the increased contention for the locks protecting these resources. Recently, several enhancements were made to the scheduler to reduce the contention for such variables and so improving scalability. The concept of *root domains* was introduced by Gregory Haskins for this purpose.

First, let's briefly introduce *cpusets*. Cpusets provide a mechanism to partition CPUs into a subset that is used by a process or a group of processes. Several cpusets could overlap, on the other hand, a cpuset is called exclusive if no other contains overlapping CPUs. Each exclusive cpuset defines an isolated domain of CPUs partitioned from other cpusets or CPUs. Whenever a cpuset is created, a root domain has to be created and attached to the one, so root domain is a way to attach all the informations describing a cpuset to the cpuset itself.

`struct root_domain` is defined in `kernel/sched/sched.h` and the most relevant field are shown in Listing 1.2.

Listing 1.2: `struct root_domain`

```
struct root_domain {
    atomic_t refcount;
    atomic_t rto_count;
    cpumask_t span;
    cpumask_t online;
    cpumask_t rto_mask;
    ...
    struct cpupri cpupri;
};
```

Root domains are so used to narrow the scope of the global variables to per-domain variables. Whenever an exclusive cpuset is created, a new root domain object is created with information from the member CPUs. By default, a single high-level root domain is created with all CPUs as members. All real-time scheduling decisions are made only within the scope of a root domain.

As we can see, the concept of root domain is the equivalent of scheduling domain inside the real-time scheduler part.

1.2.7 CPU priority management

CPU Priority Management is an infrastructure also introduced by Gregory Haskins to make task migration decisions efficient. This code tracks the priority of every CPU in the root domain. Every CPU can be in any one of the following states: INVALID, IDLE, NORMAL, RT1, ...RT99. The system maintains this state in a two-dimensional bitmap: one dimension for the different priority levels and the second for the CPUs in that priority level. CPU priority means the value in `rq->rt.highest_prio.curr`, that is, the priority of the highest priority task queued on that CPU runqueue. This is implemented using two arrays, as shown in Listing 1.3.

Listing 1.3: `struct cpupri`

```

struct cpupri_vec {
    atomic_t count;
    cpumask_var_t mask;
};

struct cpupri {
    struct cpupri_vec pri_to_cpu[CPUPRI_NR_PRIORITIES];
    int cpu_to_pri[NR_CPUS];
};

```

The field `pri_to_cpu` yields information about all the CPUs of a cpuset that are in a particular priority level. This is encapsulated in `struct cpupri_vec`.

The field `cpu_to_pri` indicates the priority of a CPU.

The `struct cpupri` is scoped at the root domain level, so every exclusive cpuset has its own `cpupri` data value.

The CPU Priority Management infrastructure is used to find a CPU to which to push a task, as shown in 1.4.

Listing 1.4: `cpupri_find` function

```

int cpupri_find(struct cpupri *cp, struct task_struct *p,

```



```

    struct cpumask *lowest_mask)
{
    int idx = 0;
    int task_pri = convert_prio(p->prio);

    if (task_pri >= MAX_RT_PRIO)
        return 0;

    for (idx = 0; idx < task_pri; idx++) {
        struct cpupri_vec *vec = &cp->pri_to_cpu[idx];
        int skip = 0;

        if (!atomic_read(&(vec)->count))
            skip = 1;
        smp_rmb();

        if (skip)
            continue;
        if (cpumask_any_and(&p->cpus_allowed, vec->mask) >= nr_cpu_ids)
            continue;
        if (lowest_mask) {
            cpumask_and(lowest_mask, &p->cpus_allowed, vec->mask);
            if (cpumask_any(lowest_mask) >= nr_cpu_ids)
                continue;
        }
        return 1;
    }

    return 0;
}

```

If a priority level is non-empty and lower than the priority of the task being pushed, the `lowest_mask` is set to the mask corresponding to the priority level selected. This mask is then used by the push algorithm to compute the best CPU to which to push the task, based on affinity, topology and cache characteristics.

1.2.8 Details of the *Push* scheduling algorithm

As discussed before, when a low priority real-time task gets preempted by a higher one or when a task is woken up on a runqueue that already has a higher priority task running on it, the scheduler needs to search for a suitable runqueue for the task. This operation of searching a runqueue and transferring one of its tasks to another runqueue is called pushing a task.

The `push_rt_task()` algorithm looks at the highest priority non-running runnable real-time task on the runqueue of the CPU calling the operation itself and considers all the others runqueues to find a CPU where it can run. It searches for a runqueue that is of lower priority, that is, one where the currently running task can be preempted by the task is being pushed.

The CPU Priority Management mechanism, detailed in Section 1.2.7, is used to find a mask of CPUs that have the lowest priority runqueues. It is important to select only the best CPU from among all the candidates. The algorithm gives the highest priority to the CPU on which the task last executed, as it is likely to be cache-hot in that location. If that is not possible, the scheduling domain map is considered to find a CPU that is logically closest to the last CPU. If this too fails, a CPU is selected at random from the mask.

The push operation is performed until a real-time task fails to be migrated or there are no more tasks to be pushed. Because the algorithm always selects the highest non-running task for pushing, the assumption is that, if it cannot migrate it, then most likely the lower real-time tasks cannot be migrated either and the search is aborted. No lock is taken when scanning for the lowest priority runqueue. When the target runqueue is found, only the lock of that runqueue is taken, after which a check is made to verify whether it is still a candidate to which to push the task, as the target runqueue might have been modified by a parallel scheduling operation on another CPU. If not, the search is repeated for a maximum of three tries, after which it is aborted.

1.2.9 Details of the *Pull* scheduling algorithm

The `pull_rt_task()` algorithm looks at all the overloaded runqueues in a root domain and checks whether they have a non runnable real-time task that can run on the runqueue of the CPU calling the function, namely the target runqueue. The task can run on the target runqueue if the target CPU bit is set in the `cpumask` structure of the eligible task. Moreover, the eligible task priority has to be higher than that of the task the target runqueue is

about to schedule. If so, the task is queued on the target runqueue. This search aborts only after scanning all the overloaded runqueues in the root domain. Thus, the pull operation may pull more than one task to the target runqueue.

As in the push operation, the pull selects a candidate task in the first pass, and then performs the actual pull in the second pass, so there is a possibility that the selected task is no longer a candidate, due to another parallel scheduling operation executed in the meanwhile. To avoid this race the pull operation continues to pull tasks even if the operation fails. In the worst case, this might lead to a number of tasks being pulled to the target runqueue which would later get pushed away to other CPUs, leading to the so called *task bouncing* phenomenon.

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

During the last years, research institutions and independent developers have proposed several real-time extensions to the Linux kernel, in order to address the deficiencies of SCHED_FIFO and SCHED_RR scheduling classes. In this section we present a brief description of the more interesting alternatives.

1.3.1 RTLinux, RTAI and Xenomai

RTLinux is a patch developed at *Finite State Machine Labs* (FSMLabs) to add real-time features to the standard Linux kernel [33]. 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 idea 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 practice, 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*” [30]. 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 [11]. The ADEOS nanokernel implements a pipeline scheme into which every domain (OS) has an entry with a predefined priority. RTAI 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 [16] 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 [20] 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.

1.3.2 PREEMPT_RT

The CONFIG_PREEMPT_RT [17] 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 interrupts 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 priority task is preempted by a medium priority task while holding the lock, the high priority task will have to wait for the medium priority task to complete, that is, for a possibly long (and 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 [20].

1.3.3 OCERA

OCERA [26], 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 techniques for build embedded systems.

A real-time scheduler for Linux 2.4 has been developed within this project, and it is available as open source code [3], [29], [31]. To minimize the mod-

ifications 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.

1.3.4 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).

1.3.5 FRESCOR

FRESCOR [15] is a consortium research project funded in part by the European Union's Sixth Framework Programme [13]. 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.

1.3.6 LITMUS^{RT}

The LITMUS^{RT} [21] 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).

1.4 EDF and CBS theory

In this section we are going to detail one fundamental real-time scheduling algorithm. As we will see in Section 1.5, an implementation of this algorithm is already available in Linux as a new scheduling class.

In order to understand this algorithm, we first present a brief discussion of the theory behind that. 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 simplifying 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.

1.4.1 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 arbitrarily*). The scheduling goes on this way until instant 24 (*hyperperiod*, least common multiple of tasks periods), then repeats the same.

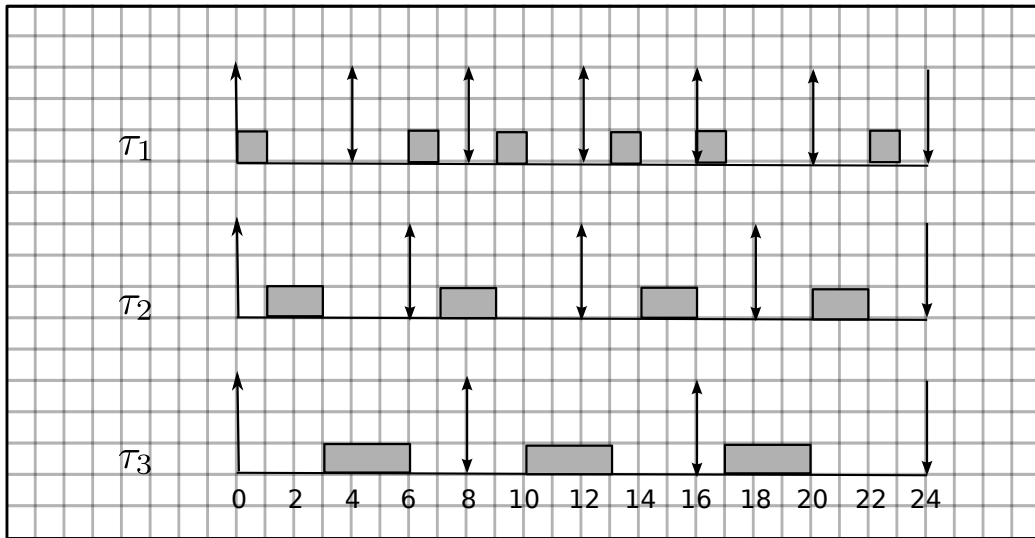


Figure 1.3: An EDF scheduling example.

Last thing to say is about schedulability bound with EDF:

- **Theorem** [22]: 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 $U = \frac{23}{24} \leq 1$.

1.4.2 Constant Bandwidth Server

In Section 1.4.1 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 (that is: a *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 at 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 idea 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. At 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.

1.4.3 EDF scheduling on SMP systems

In this thesis we will consider the problem of scheduling soft real-time tasks on a Symmetric Multi Processor (*SMP*) platform, made up by M identical processors (or cores) with constant speed.

On a multi-core platform, there are three different approaches to schedule a task set:

partitioned-EDF tasks are statically assigned to processors and those on each processor are scheduled on an EDF basis. Tasks are so pinned to a specific runqueue without the possibility of migrate between those. Therefore, in an M processor system we have M task sets independently scheduled. The main advantage of this approach is its simplicity, as a multiprocessor scheduling problem is reduced to M uniprocessor ones. Furthermore, tasks experience no overhead, since there aren't migrations. On the contrary, drawbacks of P-EDF are the complexity to find an optimal assignment of tasks to processors (which is NP-hard) and the impossibility to schedule some particular task sets that are schedulable only if task sets are not partitioned [6].

global-EDF jobs are inserted in a global deadline-ordered ready queue, and on a instant by instant basis the available processors are allocated to the nearest deadline jobs in the ready queue.

hybrid-EDF 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, as if each cluster be constituted by an independent system for scheduling purposes.

No variant of EDF is optimal, so deadline misses can occur under each EDF variant in a feasible systems⁹. It has been shown, however, that deadline tardiness under G-EDF is bounded in systems, which, as we said, is sufficient for many soft real-time applications [9, 32].

Under the H-GDF 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.

1.5 The `SCHED_DEADLINE` scheduling class

`SCHED_DEADLINE` [14] is a 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.¹⁰ in the context of the EU-Funded project ACTORS¹¹.

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 desir-

⁹Systems with total utilization at most the number of processors

¹⁰<http://www.evidence.eu.com>

¹¹<http://www.actors-project.eu/>

able 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., AKAtch) 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 as well.
- 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.

1.5.1 Main Features

`SCHED_DEADLINE`¹² implements the Earliest Deadline First algorithm and uses the Constant Bandwidth Server to provide *bandwidth isolation*¹³ among tasks. The scheduling policy does not make any restrictive assumption about the characteristics of tasks: it can handle periodic, sporadic or aperiodic tasks.

¹²The new `kernel/sched/dl.c` file contains the scheduling policy core.

¹³Different 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.

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 as not to 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 the 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 1.4 shows the Linux scheduling framework with SCHED_DEADLINE added.

1.5.3 Current Multiprocessor Scheduling Support

As we have seen in Section 1.1.2, in 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.

SCHED_DEADLINE developers has initially chose to implement the P-EDF solution, where no dynamic processes migration can take place, unless we change the task affinity.

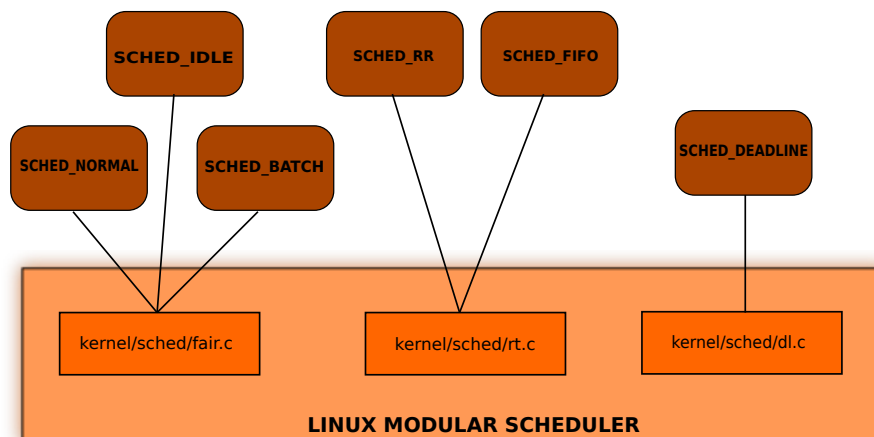


Figure 1.4: The Linux modular scheduling framework with `SCED_DEADLINE`.

Recently, Juri Lelli, the current maintainer of `SCED_DEADLINE` project, has extended its implementation to allow a G-EDF and a H-EDF scheduling schemes. At the time of writing, in `SCED_DEADLINE` newer version, we found not only the same distributed runqueue approach that all other scheduling classes follow, but also the push and pull algorithms to balance the load over all CPUs in the system. Obviously, here the migrations are done comparing the tasks deadline.

The goal of this design is to approximate as much as possible the G-EDF rule: “on an M CPUs system, the M earliest deadline ready tasks run on the CPUs”. We use the term approximate because it’s clear that there may be some intervals in which the above rule may be violated: in fact, scheduler can migrate tasks only when they are woken up or when their relative deadline changes, in a similar manner as we have seen in 1.2.4 for `SCED_RT` scheduling class tasks. In other words, the scheduler uses only local informations to impose a schedule, while occasionally relying on the push and pull mechanisms to achieve a global balancing.

Compared to a global scheduling policy with a single system-wide runqueue, this solution has the advantage of a better scalability as the number of underlying cores increases. In fact, we have to keep in mind that, on a M processors SMP system, we can have up to M scheduler instances executing

at the same time, that compete to acquire the lock on the single runqueue.

Now, let us briefly discuss the data structures and the algorithms behind the SCHED_DEADLINE support to multi-core environments.

The concept of root domain is used here as in SCHED_RT, but the `struct root_domain` is extended to manage the deadline tasks, so we can find the following additional fields:

Listing 1.5: `struct root_domain` extended

```
struct root_domain {
    <same fields as above>
    ...
    cpumask_var_t dlo_mask;
    atomic_t dlo_count;
    ...
    struct cpudl cpudl;
};
```

The field `dlo_mask` shows which CPUs are overloaded and `dlo_count` keeps count of those. The remaining field, `struct cpudl`, is fundamental to speed up the push mechanism. In the current implementation, that data structure is a max-heap that keeps the deadline of the earliest deadline task in all the runqueue.

As we will see in the remaining part of this document, the main goal of this thesis is to design and develop more efficient data structures to speed up the migration algorithms.

To implement the tasks migration mechanism, SCHED_DEADLINE also uses some particular fields on his runqueue structure, as we can see in Listing 1.6.

Listing 1.6: `struct dl_rq`

```
struct dl_rq {
    struct rb_root;
    struct rb_node *rb_leftmost;
    unsigned long dl_nr_running;

#ifdef CONFIG_SMP
    struct {
        u64 curr;
    };
};
```



```

    u64 next;
} earliest_dl;

unsigned long dl_nr_migratory;
unsigned long dl_nr_total;
int overladed;

struct rb_root pushable_dl_tasks_root;
struct rb_node *pushable_dl_tasks_leftmost;
#endif
...
};

```

The struct `dl_rq` is the place where we store task accounting information to manage overloading and migrations. Among these the most important fields are:

struct earliest_dl a cache for the two earliest deadline task enqueued in the runqueue, to speed up push and pull decisions.

dl_nr_migratory the number of deadline tasks that can migrate.

dl_nr_total total number of deadline tasks queued.

pushable_dl_tasks_root the root of a red-black tree where pushable deadline tasks are enqueued.

pushable_tasks_leftmost pointer to the earliest deadline pushable task.

1.5.4 SCHED_DEADLINE Push implementation

Now, let us discuss in great detail the push algorithm implemented in `SCHED_DEADLINE` scheduling class. In Listing 1.7 we can see the main push mechanism function: `push_dl_task`.

Listing 1.7: *Push function*

```

static int push_dl_task {
    struct task_struct *next_task;
    struct rq *later_rq;

    if (!rq->dl.overloaded)

```

```

    return 0;

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

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

    /*
     * If next_task preempts rq->curr, and rq->curr
     * can move away, it makes sense to just reschedule
     * without going further in pushing next_task.
     */
    if (dl_task(rq->curr) &&
        dl_time_before(next_task->dl.deadline, rq->curr->dl.deadline) &&
        rq->curr->dl.nr_cpus_allowed > 1) {
        resched_task(rq->curr);
        return 0;
    }

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

    /* Will lock the rq it'll find */
    later_rq = find_lock_later_rq(next_task, rq);
    if (!later_rq) {
        struct task_struct *task;

        /*
         * We must check all this again, since
         * find_lock_later_rq releases rq->lock and it is
         * then possible that next_task has migrated.
         */
        task = pick_next_pushable_dl_task(rq);
        if (task_cpu(next_task) == rq->cpu && task == next_task) {
            /*
             * The task is still there. We don't try
             * again, some other cpu will pull it when ready.
             */
            dequeue_pushable_dl_task(rq, next_task);
            goto out;
        }

        if (!task)
            /* No more tasks */
            goto out;
    }

```

```

    put_task_struct(next_task);
    next_task = task;
    goto retry;
}

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

resched_task(later_rq, next_task, 0);

double_unlock_balance(rq, later_rq);

out:
    put_task_struct(next_task);

    return 1;
};

```

The *push* function first checks the overloaded flag to see if there are deadline tasks to push away, then pick from the pushable rbtree the task to try to push next. At this time, `find_lock_later_rq` find and lock a runqueue where the task can immediately run, that is, the pushable task will preempt the task currently executing on the target runqueue. If such a runqueue is found then the actual migration is accomplished, otherwise the function just retries or exits.

The `find_lock_later_rq` code is presented in Listing 1.8.

Listing 1.8: *pick_next_pushable_dl_task* function

```

static struct rq *find_lock_later_rq(struct task_struct *task, struct rq *rq)
{
    struct rq *later_rq = NULL;
    int tries;
    int cpu;

    for(tries = 0; tries < DL_MAX_TRIES; tries++) {
        cpu = find_later_rq(task);

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

        later_rq = cpu_rq(cpu);
    }
}

```

```

/* Retry if something changed. */
if (double_lock_balance(rq, later_rq)) {
    if (unlikely(task_rq(task) != rq ||
        !cpumask_test_cpu(later_rq->cpu,
            &task->cpus_allowed) ||
        task_running(rq, task) ||
        !task->on_rq)) {
        double_unlock_balance(rq, later_rq);
        later_rq = NULL;
        break;
    }
}

/*
 * If the runqueue we found has no -deadline task, or
 * its earliest one has a later deadline than our
 * task, the rq is a good one.
 */
if(!later_rq->dl.dl_nr_running ||
    dl_time_before(task->dl.deadline,
        later_rq->dl.earliest_dl.curr))
    break;

/* Otherwise we try again */
double_unlock_balance(rq, later_rq);
later_rq = NULL;
}

return later_rq;
}

```

This function tries up to `DL_MAX_TRIES` (that is, three times in the current implementation) times to find a suitable runqueue to push the task away. It only acquires a double lock, one on the source and the other on the destination runqueues if it succeeds in its work. A check is performed immediately after that the locks are acquired to see if a parallel scheduling operation makes the target runqueue no more eligible to immediately run the task to migrate. The very core of all mechanism is inside the `find_later_rq` function. Here we show only the relevant part:

Listing 1.9: `find_later_rq` function

```

static int find_later_rq(struct task_struct *task)
{
    struct sched_domain *sd;

```

```

struct cpumask *later_mask = __get_cpu_var(local_cpu_mask_dl);
int this_cpu = smp_processor_id();
int best_cpu, cpu = task_cpu(task);

/* Make sure the mask is initialized first */
if (unlikely(!later_mask))
    return -1;

if (task->dl.nr_cpus_allowed == 1)
    return -1;

best_cpu = cpudl_find(&task_rq(task)->rd->cpudl,
                    task_rq(task)->rd->dlo_mask,
                    task, later_mask);
if (best_cpu == -1)
    return -1;

...

return best_cpu;
}

```

1.5.5 Max-heap `cpudl` data structure for push operation

As we have seen, the function `find_later_rq` relies on the `cpudl` data structure to efficiently find a target runqueue (that is, a target CPU) where to push the task.

In the current `SCHED_DEADLINE` implementation the `cpudl` data structure is a max-heap that stores the deadline value of the tasks currently executing on a CPU. We can see an example of such a structure in Figure 1.5 on page 35 where a 4-CPU system is represented. In the above Figure the `cpudl` data structure is simply represented as an ordered queue, since we will see that many possible solutions are available for the implementation of such a structure.

The `cpudl` data structure is managed through a simple API made up of two function, as we can see in Listing 1.10.

Listing 1.10: `cpudl` API

```
int cpudl_find(struct cpudl *cp, struct cpumask *dlo_mask,
              struct task_struct *p, struct cpumask *later_mask);
void cpudl_set(struct cpudl *cp, int cpu, u64 dl, int is_valid);
```

The *find* operation is called when a scheduler instance, running on a CPU, has to migrate a task and needs to know where it can push one.

The *set* operation is called when a scheduler instance, running on a CPU, has to update the `cpudl` data structure to reflect a change in the underlying runqueue status.

To implement a scheduling policy as close as possible to *G-EDF*, the `cpudl` data structure keeps also track of the free CPUs (that is, a CPU with no deadline tasks enqueued in its runqueue). When a CPU needs to know where to push a task, `cpudl_find` first looks into a proper CPU bitmask where all free CPUs has an associated cleared bit. If it is possible to find at least one free CPU, we don't have to search in the max-heap and we can immediately return the CPU index founded.

Now, let us focus on the `cpudl_find` and `cpudl_set` parameters. Regarding the former operation, we have the following parameters:

cp same as above.

dlo mask not used in current version.

p a pointer to the task to migrate. We use this pointer to read the CPU affinity of the task in its `task_struct`. In this way, `cpudl_find` can always returns an eligilble CPU index where task `p` is allowed to run.

later_mask a pointer to a CPU bitmask where `cpudl_find` can set all bits related to CPUs eligible for the migration. In particular, this mask is used when there are more than one free CPUs and `cpudl_find` lets the caller choose which CPU is best.

Regarding the latter one, we have to specify the following parameters:

cp a pointer to the instance of the `cpudl` data structure. In fact, there are as many different instances of `cpudl` data structures as the number of root domains.

cpu the index of the CPU that is calling the function.

dl the new deadline value of the currently running task on `cpu`.

is_valid a flag to indicate if there is at least one

deadline task enqueued in the runqueue.

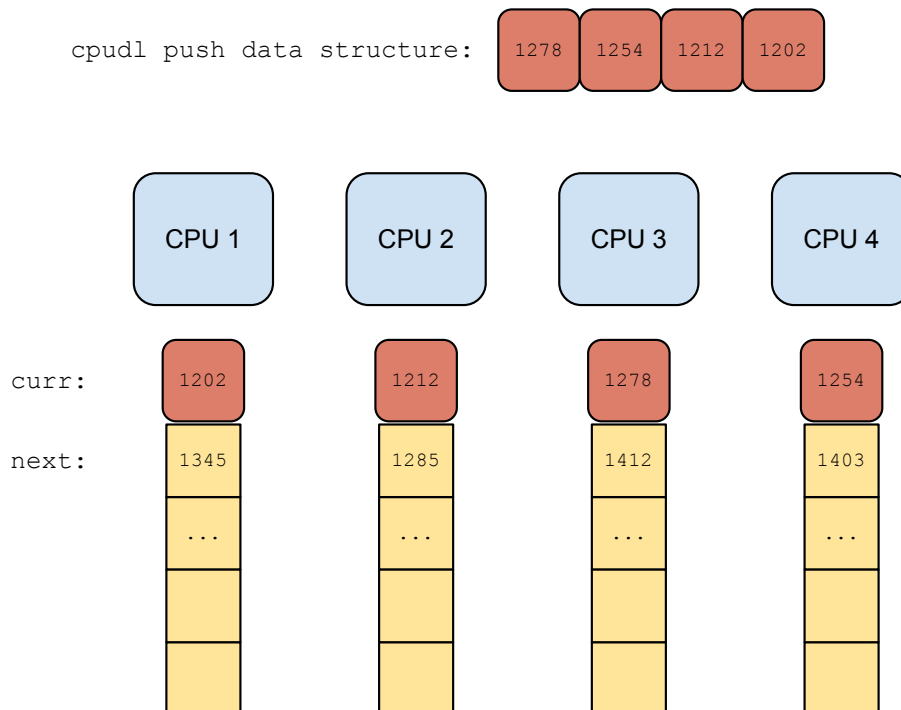


Figure 1.5: `cpudl` structure for *push* operation.

1.5.6 SCHED DEADLINE Pull implementation

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 immediately. If found, this function performs a migration, otherwise it continues or just exits if there are no more runqueue to consider. So, we can state that the main goal of both push and pull operations is to perform a preemption in the target runqueue.

The pull operation is implemented in the `pull_dl_task` function, presented in Listing 1.11. For brevity's sake we remove all the comments from the code.

Listing 1.11: *pull_dl_task* 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;
    u64 dmin = LONG_MAX;

    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);

        if (this_rq->dl.dl_nr_running &&
            dl_time_before(this_rq->dl.earliest_dl.curr,
                          src_rq->dl.earliest_dl.next))
            continue;

        double_lock_balance(this_rq, src_rq);

        if (src_rq->dl.dl_nr_running <= 1)
            goto skip;

        p = pick_next_earliest_dl_task(src_rq, this_cpu);

        if (p && dl_time_before(p->dl.deadline, dmin) &&
            (!this_rq->dl.dl_nr_running ||
             dl_time_before(p->dl.deadline,
                           this_rq->dl.earliest_dl.curr))) {
            WARN_ON(p == src_rq->curr);
            WARN_ON(!p->on_rq);

            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);
        dmin = p->dl.deadline;
    }
skip:
    double_unlock_balance(this_rq, src_rq);
}

return ret;
}

```

The key difference between the *pull* operation and the *push* function, is that inside pull we have to check every single runqueue in order to find tasks to pull. In other words, in the current implementation, there isn't available an analogous data structure like the `cpudl` one for the *push* operation. On a large SMP system, with a considerable number of cores, this can lead to an unsustainable latency to perform the *pull* operation.

We will see in Chapter 3 how we have addressed this problem.

1.5.7 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 (while this feature is not available in `SCHED_RT` scheduling class, as we seen in Section 1.2.1). This means that 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 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.

1.5.8 Usage and Tasks 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_dl_period.us` and `/proc/sys/kernel/sched_dl_runtime.us` files. The quantity

$$\frac{\text{sched_dl_runtime.us}}{\text{sched_dl_period.us}}$$

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

Otherwise, it is possible to disable SCHED_DEADLINE bandwidth control echoing the value `-1` to in `/proc/sys/kernel/sched_dl_runtime.us`. 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_param2`¹⁴ has been implemented. It allows to assign or modify the scheduling parameters de-

¹⁴defined in `include/linux/sched.h`

scribed above (i.e., `sched_dl_runtime` and `sched_dl_period`) for tasks running with `SCHED_DEADLINE` policy.

The `struct sched_param2` implementation can be seen in Listing 1.12.

Listing 1.12: `struct sched_param2`

```
struct sched_param2 {
    int sched_priority;
    unsigned int sched_flags;
    u64 sched_runtime;
    u64 sched_deadline;
    u64 sched_period;
};
```

The syscall has the following prototype:

Listing 1.13: `sched_setscheduler2` syscall

```
int sched_setscheduler2(struct task_struct *p, int policy,
    const struct sched_param2 *param);
```

For the sake of consistency, also

Listing 1.14: `sched_setparam2` and `sched_getparam2` syscalls

```
int sched_setparam2(pid_t pid, struct sched_param2 *param);
int sched_getparam2(pid_t pid, struct sched_param2 *param);
```

have been implemented.

Chapter 2

Synchronization mechanisms analysis

As we stated in the previous chapter, the main goal of this thesis is to design and develop new migration mechanisms that scale well while the number of underlying cores increases. So, we can't leave aside a detailed description of the various synchronization mechanisms used to ensure a correct interaction between multiple threads of execution. In particular, we are going to detail the facilities that the Linux kernel provides to developers.

After that, we will explain a novel (and widely applicable) framework to efficiently manage concurrent accesses to a shared data structures, called *flat combining*.

2.1 Kernel locking techniques

The fundamental issue surrounding locking is the need to provide mutual exclusion in certain code paths in the kernel. These code paths, called *critical sections*, require some combination of concurrency or re-entrancy protection and proper ordering with respect to other events. The typical result without proper locking is called a *race condition*: the output is dependent on the sequence of events. To avoid race conditions we need to rely on locking. The Linux kernel provides a family of locking primitives that developers can use

to write safe and efficient code.

2.1.1 SMP and UP Kernel

Depending on the configuration used to compile the kernel, Linux can be configured to be used in a uniprocessor (*UP*) or in a multiprocessor (*SMP*) environment. Some locking issues arises only in a *SMP* kernel, where we have real parallelism, that is, more than one instructions are executed at the exact same time. But even in a *UP* kernel we may have some locking issues: if it is compiled with preemption enabled, a kernel can preempt itself, thus leading to the need of locking usage.

Linux locking primitives are written in order to ensure proper synchronization with all kinds of kernel, thanks to the conditional compilation enabled by two macros:

- `CONFIG_SMP` to enable kernel *SMP* support
- `CONFIG_PREEMPT` to enable kernel preemption support

In the following analysis we will refer to a kernel with both two macros defined.

2.1.2 Atomic operators

Atomic operators are maybe the simplest of the approaches to kernel synchronization and thus probably the easiest to understand and use. In addition to this, they are the building blocks of the kernel's locks.

Atomic operators are operations, like add and subtract, which execute in one uninterruptible operation. There are two different subsets of atomic operations: methods that operates on integers and methods that operates on bits. For the sake of simplicity, we are going to describe only the first subset.

The most important atomic operations are listed in the following Listing2.1.

Listing 2.1: Atomic operations on integer

```
void atomic_set(atomic_t *v, int i);
int atomic_read(const atomic_t *v);
void atomic_add(int i, atomic *v);
void atomic_sub(int i, atomic_t *v);
int atomic_cmpxchg(atomic_t *v, int old, int new);
```

The above primitives work on a integer variable (encapsulated in `atomic_t` type), that extends on 32 bits on most hardware architectures. Others atomic operations for 64-bit variables are also available.

The semantics of the above operations is quite straightforward, but there is one of those that deserves a further explanation. The `atomic_cmpxchg` operation is fundamental, because it allows to realize the so called *CAS* (Compare-And-Swap) operation: the value of the memory location addressed by `v` pointer is atomically exchanged with the new value iff memory contains the `old` value. If the exchange actually takes place, `atomic_cmpxchg` returns `old` value, otherwise it returns a different value. This operation is also particular because it is the only one among the above that issues a full memory barrier. We will discuss about memory barriers in Section 2.2.

2.1.3 Spinlocks

For anything more complicated than the basic arithmetic operations, a more complete locking solutions is needed. The most common locking primitive in the kernel is the spinlock. The spinlock is a very simple single-holder lock. If a process attempts to acquire a spinlock and it is unavailable, the process will keep trying (that is: spinning) until it can acquire the lock. This simplicity leads to a small and fast lock.

An example of usage is in Listing 2.2.

Listing 2.2: Spinlock operations

```
spinlock_t lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;

spin_lock_irqsave(&lock, flags);
/* critical section */
spin_unlock_irqrestore(&lock, flags);
```

The use of `spin_lock_irqsave` will disable interrupts locally and implement the spinlock on *SMP* systems. With a call to `spin_unlock_irqrestore`, interrupts are restored to the state when the lock was acquired. All of the above spinlocks assume the data they are protecting is accessed in both interrupt handlers and normal kernel code. If that critical section is accessed only in user-context kernel code (like a system call) the variants `spin_lock()` and `spin_unlock` have to be used instead of the above.

In Linux, spinlocks are not recursive, as in other operating systems: the programmer has to carefully deal with them in order to avoid potential deadlocks.

Spinlocks should be used to lock data in situations where the lock is not held for a long time: a waiting process will spin, doing nothing, waiting for the lock to be available.

Another fundamental API provided by Linux is `spin_trylock_irqsave`: it is a non-blocking variant of `spin_lock_irqsave` that returns zero if the lock is successfully acquired, otherwise it returns a non-zero value without spin. In the subsequent chapters, we will see how this primitive can effectively be used to implement lock-free solutions for shared data structures concurrency management.

2.1.4 Semaphores

Semaphores in Linux are implemented as sleeping locks: a task that fails to acquire the semaphore due to contention is forced to sleep. Because of this, semaphores are usually used in situations where the lock-held time may be long. Conversely, since they have a non negligible overhead of putting a task to sleep and subsequently waking it up, they should not be used where the lock-hold time is short. On the other hand, a task can safely block while holding a semaphore, so they can be used to synchronize user contexts.

In Linux, semaphores are represented by a structure, `struct semaphore`, that contains:

- a pointer to a *wait queue*

- a *usage count*

The wait queue is a list of processes blocking on the semaphore, while the usage count is the number of concurrently allowed holders. If it is negative, the semaphore is unavailable and the absolute value of the usage count is the number of processes blocked on the wait usage.

The primitives used to manage a semaphore is showed in Listing 2.3.

Listing 2.3: Semaphore operations

```
void sema_init(struct semaphore *sem, int val);
int down_interruptible(struct semaphore *sem);
void down(struct semaphore *sem);
void up(struct semaphore *sem);
```

The *sema_init* simply initializes the semaphore. The *up* function is used to release the semaphore, incrementing the usage count. If the new value is greater than or equal to zero, one or more tasks on the wait queue will be woken up.

To attempt to acquire a semaphore, we have to use one among *down_interruptible* and *down* functions: the former decrements the usage count of the semaphore and, if the new value is less than zero, the calling process is added to the wait queue and blocked. If the new value is greater or equal to zero, the process obtains the semaphore and the call returns 0. If a signal is received while blocking, the call returns the `-EINTR` error code and the semaphore is not acquired. The latter performs almost the same, except that it puts the calling task into an uninterruptible sleep: a signal received by a process in such a status is ignored.

2.1.5 Reader/Writer locks

In addition to spinlocks and semaphores, Linux provides reader/writer variants that divide lock usage into two groups: reading and writing. Since it is safe for multiple threads to read data concurrently, so long as nothing modifies the data, reader/writer locks allow multiple concurrent readers but only a single writer (with no concurrent readers). If the data accesses can be

clearly divided into reading and writing patterns, especially with a greater amount of reading than writing, the reader/writer locks are to be preferred. In Listings 2.4 we provide an usage example of reader/writer spinlocks and reader/writer sempahores, respectively.

Listing 2.4: Reader/Writer Spinlocks

```

rwlck_t rw_lock = RW_LOCK_UNLOCKED;

read_lock(&rw_lock);
/* critical section (read only) */
read_unlock(&rw_lock);

write_lock(&rw_lock);
/* critical section (read and write) */
write_unlock(&rw_lock);

```

Listing 2.5: Reader/Writer Semaphores

```

struct rw_semaphore rw_sem;
init_rwsem(&rw_sem);

down_read(&rw_sem);
/* critical section (read only) */
up_read(&rw_sem);

down_write(&rw_sem);
/* critical section (write only) */
up_write(&rw_sem);

```

Use of those kind of locks, where appropriate, is an appreciable optimization.

2.2 Memory barriers

Before discussing memory barriers, we need to introduce the mechanisms that rules the interaction between CPUs and memory in a multiprocessor environment. After that, it will be clear how important memory barriers are while developing lock-free solutions in a multicore environment.

For further details about memory barriers see [23].

2.2.1 Abstract memory access model

Consider the abstract model of the system in Figure 2.1.

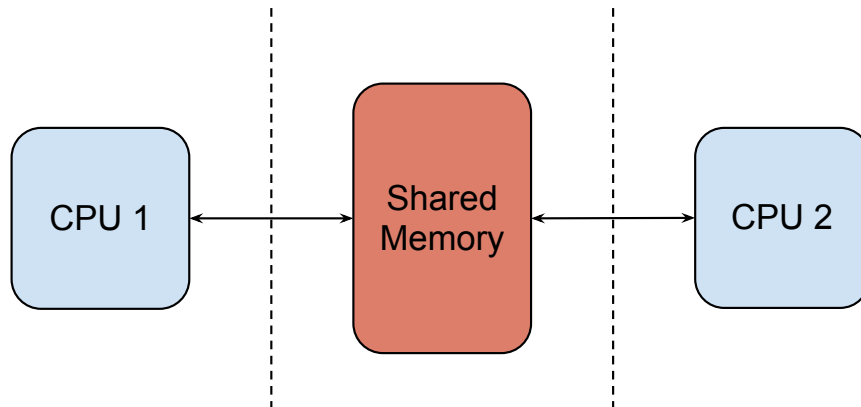


Figure 2.1: An abstract model of a multiprocessor system.

Each CPU executes a program that generates memory access operations. In the abstract CPU, memory operation ordering is very relaxed: a CPU may actually perform the memory operations in an order it likes, provided *program causality* appears to be maintained. Similarly, the compiler may also arrange the instructions it emits in any order it like, provided it does not affect the apparent operation of the program.

So, in the above diagram, the effects of the memory operations performed by a CPU are perceived by the rest of the system as the operations cross the interface between the CPU and rest of the system.

As an example, consider the sequence of events shown in Table 2.1.

CPU 1	CPU 2
{A == 1; B == 2}	
A = 3;	x = A;
B = 4;	y = B;

Table 2.1: A sequence of memory operations performed by two CPUs

The set of accesses as seen by the memory system can be arranged in 24 different combinations, some of them are showed below as examples.

```

STORE A=3, STORE B=4, x=LOAD A→3, y=LOAD B→4;
STORE A=3, STORE B=4, y=LOAD B→4, x=LOAD A→3;
STORE A=3, x=LOAD A→3, STORE B=4, y=LOAD B→4;
STORE A=3, x=LOAD A→3, y=LOAD B→2, STORE B=4;
STORE A=3, y=LOAD B→2, STORE B=4, x=LOAD A→3;
STORE A=3, y=LOAD B→2, x=LOAD A→3, STORE B=4;
STORE B=4, STORE A=3, x=LOAD A→3, y=LOAD B→4;

```

...

Since all of the above permutations are eligible, the final result can be one of the subsequent four different combinations of values:

```

x == 1, y == 2
x == 1, y == 4
x == 3, y == 2
x == 3, y == 4

```

Furthermore, the stores committed by a CPU to the memory system may not be perceived by the loads made by another CPU in the same order as the stores were committed.

As a further example, consider the sequence of events showed in Table 2.2

CPU 1	CPU 2
{A == 1, B == 2, C == 3, P == &A, Q == &C}	
B = 4;	Q = P;
P = &B;	D = *Q;

Table 2.2: Another sequence of memory operations performed by two CPUs

There is an obvious data dependency here, as the value loaded into D depends on the address retrieved from P by CPU 2. At the end of the sequence, any of the following results are possible:

(Q == &A) and (D == 1)

(Q == &B) and (D == 2)

(Q == &B) and (D == 4)

Note that CPU 2 will never try and load C into D because the CPU will load P into Q before issuing the load of *Q.

2.2.2 CPU guarantees

Since we have stated that any CPU may reorder instructions until it doesn't affect *program causality*, let's now list the minimal guarantees that a programmer may be expected from a CPU:

- On any given CPU, dependent memory accesses will be issued in order, with respect to itself. This means that for:

$$Q = P; D = *Q;$$

the CPU will issue the following memory operations:

$$Q = \text{LOAD } P, D = \text{LOAD } *Q$$

and always in that order.

- We say that loads and stores operations *overlap* if they are targeted at overlapping pieces of memory. So, overlapping loads and stores within a particular CPU will appear to be ordered within that CPU. This means that for:

$$a = *X; *X = b;$$

the CPU will only issue the following sequence of memory operations:

$$a = \text{LOAD } *X, \text{STORE } *X = b$$

And for:

$$*X = c; d = *X;$$

the CPU will only issue:

$$\text{STORE } *X = c, d = \text{LOAD } *X$$

Besides those, there are a number of things that **must** or **must not** be assumed:

- It **must not** be assumed that *independent* (that is, not overlapping) loads and stores will be issued in the order given.
- It **must** be assumed that overlapping memory accesses may be merged or discarded. This means that for:

$$*A = X; Y = *A;$$

we may get any one of the following sequences:

$$\text{STORE } *A = X; Y = \text{LOAD } *A; \text{STORE } *A = Y = X;$$

2.2.3 Behaviour and varieties of memory barriers

As seen above, independent memory operations are effectively performed in random order, this can be a problem for CPU to CPU interaction (and even for interaction with the I/O subsystem). What is required is some way to instruct the compiler and the CPU to restrict the order.

Memory barriers have been created for this purpose: they impose a perceived *partial ordering* over the memory operations on either side of the barrier.

Such enforcement is important because the CPUs can use a variety of tricks to improve performance, including reordering, deferral and combination of memory operations, speculative loads, speculative branch prediction

and various type of caching. Memory barriers are thus used to override or suppress these tricks, allowing the code to sanely control the interaction of multiple CPUs.

Memory barriers come in four basic varieties:

Write memory barriers These barriers gives the guarantee that all STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other CPUs of the system.

A write barrier is a partial ordering on stores only: it is not required to have any effects on load.

Data dependency barriers They are a weaker form of read barrier. In the case where two loads are performed such that the second depends on the result of the first (e.g.: the first load retrieves the address to which the second load will be directed), a data dependency barrier would be required to make sure that the target of the second load is updated before the address obtained by the first load is accessed.

A data dependency barrier is a partial ordering on interdependent loads only; it is not required to have any effects on stores, independent loads or overlapping loads.

Read memory barriers Those barriers are like data dependency type plus a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other CPUs of the system.

A read barrier is a partial ordering on loads only; it is not required to have any effect on stores.

General memory barriers A general memory barrier gives a guarantee that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other CPUs of the system.

A general memory barrier implies both read and write memory barriers, and so can substitute for either.

There are also a couple of implicit varieties:

LOCK operations This acts as one-way permeable barrier. It guarantees that all memory operations after the LOCK operation will appear to happen after the LOCK operation with respect to the other components of the system.

UNLOCK operations This also acts as a one-way permeable barrier. It guarantees that all memory operations before the UNLOCK operation will appear to happen before the UNLOCK operation with respect to the other components of the system.

LOCK and UNLOCK operations are guaranteed to appear with respect to each other strictly in the order specified.

It is important to note that these are *minimum* guarantees that barriers provide. Different architectures may give more substantial guarantees, but they may not be relied upon outside of architecture specific code in Linux.

2.2.4 SMP barriers pairing

It is important to point out that there are certain things that the Linux kernel memory barriers does not guarantee:

- There is no guarantee that any of the memory accesses specified before a memory barrier will be complete by the completion of a memory barrier instruction: the barrier can be considered to draw a line in that CPU's access queue that accesses of the appropriate type may not cross.
- There is no guarantee that issuing a memory barrier on one CPU will have any direct effect on another CPU or any other hardware in the system. The indirect effect will be the order in which the second CPU sees the effects of the first CPU's accesses occur.

- There is no guarantee that a CPU will see the correct order of effects from a second CPU’s accesses, even if the second CPU uses a memory barrier, unless the first CPU also uses a matching memory barrier.

Starting from the last two points, we can understand that, when dealing with CPU to CPU interactions, certain types of memory barrier should always be paired.

A write barrier should always be paired with a data dependency barrier or read barrier, though a general barrier would also be viable. Similarly, a read barrier or a data dependency barrier should always be paired with at least a write barrier, though, again, a general barrier is viable.

An example of such pairing is the sequence of events reported in Figure 2.2.

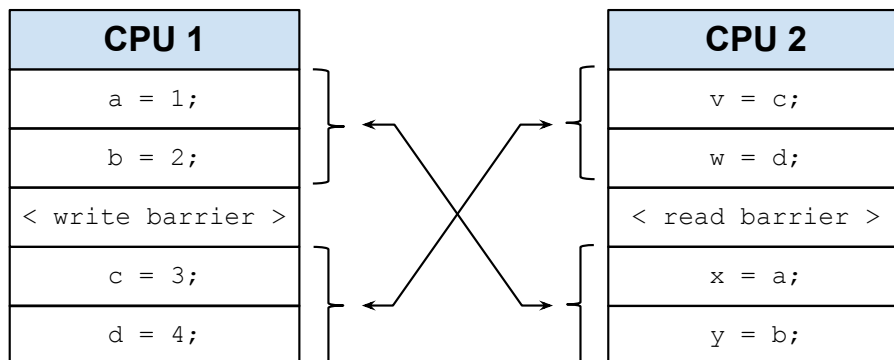


Figure 2.2: A sequence of memory operations where SMP barrier pairing is required.

Note that the stores before the write barrier would normally be expected to “match” the loads after the read barrier or the data dependency barrier, and vice versa.

2.2.5 Explicit Linux kernel barriers

The Linux kernel has a variety of different barriers that act at different levels:

- Compiler barriers: Linux has an explicit compiler barrier function that prevents the compiler from moving the memory accesses either side of it to the other side:

```
barrier()
```

This is a general barrier. The compiler barrier has no direct effect on the CPU, which may then reorder things however it wishes.

- CPU memory barriers: Linux has eight basic CPU memory barriers, as we can see in Table 2.3.

Type	Mandatory	SMP Conditional
General	<code>mb()</code>	<code>smp_mb()</code>
Write	<code>wmb()</code>	<code>smp_wmb()</code>
Read	<code>rmb()</code>	<code>smp_rmb()</code>
Data Dependency	<code>read_barrier_depends()</code>	<code>smp_read_barrier_depends()</code>

Table 2.3: Linux kernel memory barriers

All memory barriers, except the data dependency barriers imply a compiler barrier.

SMP memory barriers are reduced to compiler barriers on uniprocessor compiled systems because it is assumed that a CPU will appear to be self-consistent, and will order overlapping accesses correctly with respect to itself. So, SMP memory barriers must be used to control the ordering of references to shared memory on SMP systems, though the use of locking instead is sufficient.

Mandatory barriers should not be used to control SMP effects, since mandatory barriers unnecessarily impose overhead on UP systems.

2.2.6 Implicit kernel memory barriers

Some of the other functions in the Linux kernel imply memory barriers, amongst which are locking and scheduling functions.

It is important to point out that all the atomic operations that modify some state in memory and return information about the state (old or new) imply an SMP-conditional general memory barrier (that is: a call to `smp_mb()`) on each side of the actual operation. Among these operations we find `atomic_cmpxchg`, explained in Section 2.1.2.

2.3 Flat combining

Flat combining [18] is a new synchronization paradigm recently introduced by D. Hendler, I. Incze, N. Shavit and M. Tzafrir, that aims at reducing the synchronization overhead while accessing a shared data structure with multiple threads of execution.

The idea behind *Flat combining* is to have a given sequential data structure, named D , protected by a lock and have an associated dynamic publication list of a size proportional to the number of threads that are concurrently accessing it. Each thread accessing D for the first time adds a thread-local publication record to the list, and publishes all its successive accesses or modifications requests using a write to the request field of its publication record. In each access, after writing its request, it checks if the shared lock is free, and if so attempts to acquire it using a *CAS* (Compare-And-Set) operation. A thread that successfully acquires the lock becomes a *combiner*:

- it scans the list, collecting pending requests;
- applies the *combined requests* to D ;
- writes the results back to the threads' request fields in the associated publication records;
- finally, it releases the lock.

Otherwise, a thread that detects that some other thread already owns the lock, spins on its record, waiting for the owner to return a response in the request field, at which point it knows the published request has been applied to D . Once in a while, a combining thread will perform a cleanup operation on the publication list. During this cleanup it will remove records not recently used, so as to shorten the length of the combining traversals. Thus, in each repeated access request, if a thread has no active publication record, it will use it, and if not, it will create a new record and insert it into the list.

We can assert that flat combining is a concurrency management framework that can be adapted to many sequential data structures.

Unfortunately, not all the data structures are suited to be managed with this framework: the authors say that any data structure such that k operations on it, each taking time δ , can be combined and then applied in time *less than* $k \cdot \delta$, is a valid candidate to benefit from flat combining. So, as an example, most kind of search trees do not fit the above formula.

Furthermore, even in beneficially combinable structures, the ones that have high levels of mutation on the data structure will be rapidly beaten in performance by a finely-grained lock implementation.

Finally, we have to consider that such implementation introduces an asynchronous programming pattern: a thread that want to issue a sort of *find* operation on the data structure may have to wait until a combiner thread return the searched value in his publication record.

Chapter 3

New solutions for task migration

In this chapter we will explain several solutions designed to improve the scalability of the task migration algorithms. Our analysis will be related to new data structures and new concurrency management solutions.

3.1 Skip list

Common abstract data types like ordered lists are usually implemented through a binary tree or through a sort of balanced tree. The former is simple to develop and maintains good performance except when some particular sequences of operations are performed on it. An example of such a sequence is the inserting of elements in order: in such a scenario the tree becomes a degenerated data structure that has very poor performance. The latter has a similar behaviour but, with a more complicated algorithm, it tries to maintain certain balance conditions to ensure good performance. Obviously, we have to pay this benefit with a certain overhead that affects all operations performed on the self-balanced tree. It is possible to observe that the number of “bad” sequences are low: so, if it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input sequence. Unfortunately, in most cases, queries are answered

“online”, so randomly permuting the input is impractical.

Skip lists are a probabilistic alternative to balanced trees: they are balanced by consulting a random number generator. Although skip lists have bad worst-case performance, no input sequence consistently produces the worst-case performance, as observed in [27].

3.1.1 Skip List structure and asymptotic complexity

A skip list is capable to store a sorted list of items using a hierarchy of linked lists that connect increasingly (bottom-up) sparse subsequences of the items. An example of its structure is visible in Figure 3.1.

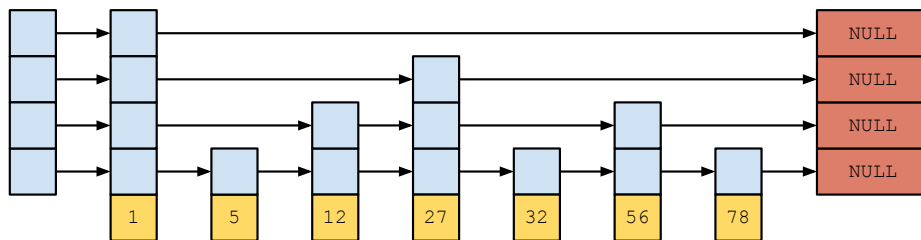


Figure 3.1: An example skip list.

Each link of the sparser lists skips over many items of the full list in one step, hence the structure’s name. These forward links may be added in a randomized way with some kind of probability distribution, typically the geometric one. Skip lists present the following operations complexity:

- Insert $\mathcal{O}(\log n)$
- Search $\mathcal{O}(\log n)$
- Delete $\mathcal{O}(\log n)$

where n is the number of items stored in the list. A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer contains extra pointers that permit to skip over intermediate nodes:

an element in layer i appears in layer $i+1$ with some fixed probability p (commonly used values are $\frac{1}{2}$ and $\frac{1}{4}$). On average, each element appears in $\frac{1}{(1-p)}$, and the tallest element (usually a special head element at the front of the skip list) in $\log_{\frac{1}{p}} n$ lists.

A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is smaller than the target. If the current element is equal to the target, it has been found. Otherwise, if the current element is greater than the target, or the search reaches the end of the linked list, the procedure is repeated after moving down vertically to the next lower list. The expected number of steps in each linked list is at most $\frac{1}{p}$.

Therefore, the total *expected* cost of a search is $\log_{\frac{1}{p}} n$, that is, as we stated above, logarithmic. Skip lists also offer the possibility to trade search costs against storage costs by choosing different values of p .

3.1.2 cpud1 skip list implementation

In this section we will present an implementation of a skip list tailored to be used in SCHED_DEADLINE migration mechanism.

The data structure has to hold the deadline value of the tasks currently executing on the CPUs (to speed-up *push* algorithm decisions) and the deadline value of the next tasks currently enqueued on the CPUs' runqueue (to speed-up *pull* algorithm decisions).

We have already seen in Section 1.5.5 the API used to cope with a certain cpud1 implementation. Since we are only modifying cpud1 itself while leaving (for now) the same *push/pull* mechanism, we decided to maintain the same API.

Now, we can make two insightful observations:

- regarding the *find* operation, we can see that the callers are always interested in picking up the first element of the data structures, that is, the index of the best CPU to where to push or pull a task;
- regarding the *set* operation, we can see that the callers always indicate

the cpu index whose deadline related value has to be updated.

So, we chose the following design to improve the accesses to the data structure:

- all the lists that compose the data structure are doubly-linked. So, we can traverse the skip list both forward and backward, starting from any item;
- we allocate a set of skip list nodes, one for each CPU in the system, and we definitively pin each node to a specific CPU. Doing so, we don't have to allocate or free memory after kernel start-up;
- all the skip list items are referenced by an array of pointers, so we can address an item simply knowing the index of the associated CPU;
- when a CPU has no deadline task in its runqueue, that is, when the scheduler running on that CPU calls `cpudl_set` with `is_valid` sets to zero, we write a specific "invalid" value in the corresponding skip list node. Consequently, we detach the node from the skip list. This node will be ready for later use and it will be addressable through the array;
- when a CPU has a deadline task in its runqueue, we can recover the associated node through the array, store the new deadline value, and then insert it in the skip list;
- finally, when a scheduler instance running on a CPU needs to know which CPU is the best for task migration, we only have to read the head element of the skip list.

To guarantee the synchronization between the different scheduler instances that issues operations on `cpudl` data structure, we used a simple `spinlock`. We have to point out that the lock must be acquired only for the *set* operation: the *find* operation is always performed lock-free through these simple steps:

- we copy the pointer to the skip list head node in a local variable;
- we check if this pointer is NULL: if so, no runqueue holds a deadline task;
- otherwise we read the CPU index and we return it to the caller.

This design leads to the following asymptotic complexities:

- find $\mathcal{O}(1)$
- set $\mathcal{O}(\log(n))$

where n is equal to the number of CPUs in the system. The code for this `cpudl` implementation is reported in Appendix A.1.

3.2 Lock-free skip list

An implementation of a lock-free skip list is described in [28]. To realize such a skip list, the author starts from an insightful observation: “the distribution of levels within a skip list effects only the performance of operations, not their correctness”. So, to delete an element we simply reduce the level of that element one step at a time, until the level is equal to one. Then, we delete it from the level one linked list, which deletes the element. If we think of a level zero element as an element that has no pointers and is not in the list, we can think of the process of deletion as reducing the level of an element down to zero. The lock-free insertions works similarly: we first insert the element in the level one linked list, then build up the level of the element as appropriate.

Unfortunately, this approach can not be easily extended to a doubly-linked skip list, as needed in `SCHED_DEADLINE` to rapidly access to an item associated with a certain CPU. So, we decided to give up with lock-free skip list to focus on another concurrency management solutions.

3.3 Bitmap flat combining

Flat Combining framework has already been briefly described in Section 2.3. Here we are going to present some improvements that aim at making the framework suitable for `SCHED_DEADLINE` integration.

3.3.1 Flat combining implementation details

Recall from the previous discussion that flat combining, as its name suggests, combines multiple operations together to complete them with a single pass on the underlying data structure. To accomplish this task, the framework relies on a list of publication records, through which the threads can request operations on the data structure.

Also, the published records list is a shared data structure that needs to be protected from concurrent accesses. In the original framework design, the authors suggest to use a linked list, with some devices to reduce contention:

- the list can not be left empty: at least one publication record must always be enqueued in it. This is useful to reduce contention between the combiner and the others thread: the former always scans the list from the head, the latter adds publication records to the tail;
- the publication records have a field that indicates if the requested operation is completed. So, even if the combiner must leave a record in the list, it knows that there are no more operations to complete;
- to avoid critical races when multiple threads add records to the list, we use a *CAS* operation on the tail of the list: this prevent us to use a lock that may quickly become a performance bottleneck as the number of threads increases;
- finally, to balance the work between threads, the authors of the original paper [18] suggest to add an *aging* mechanism. In this way, every publication record has an age field that is initialized when the record is published. The combiner thread, while scanning the list, discards

the old records. The publisher thread has to periodically check if the requested operation is completed, otherwise he has to publish again the record.

The publication records list is crucial for flat combining performance. Suppose that we have to perform a set of *insert* operations on the data structure: to “combine” the operations and insert multiple values at a time, the combiner thread needs to sort the publication records list first. In this way, it is possible to insert all the values in the structure in only one pass, one value after the other.

3.3.2 `cpud1` bitmap flat combining implementation

To obtain a suitable implementation of the flat combining framework, we bring some improvements to the publication records list. Obviously, it was not possible to use the framework “as is” in `SCHED_DEADLINE`: the publication records list would have soon arisen scalability issues, both for the contention while adding new records and for the sorting of all requests prior to execute the “combined” operations. Moreover, it is not possible to use an asynchronous programming model inside push and pull operations: whenever a scheduler instance running on a CPU has to migrate a task, it needs to know immediately which runqueue to choose.

It was decided to implement the publication records list as a hierarchical bitmap.

The top layer is a 64-bits bitmap: each bit is associated to a CPU in the system. Whenever a CPU has at least one publication record active, the corresponding bit in the 64-bits bitmap is set.

The bottom layer is made of a set of 32-bits bitmap, one for each CPU. Every 32-bits bitmaps keep tracks of the records published by a CPU (more precisely, by a scheduler instance running on a CPU). So, in this implementation, every CPU can publish at most 32 operations at a time.

As in the `cpud1` skip list implementation, all the publication records are pre-allocated at kernel start-up and freed only at system shutdown: no overhead due to memory management will slow down the migration mechanism.

The main reason to use bitmaps to arrange the publication records is the speed of the functions that operate on them. In fact, almost every modern architecture provides, in its Instruction Set, a mean to know which is the first or the last bit set in such a bitmap. Since these operations are hardware-implemented, they are usually very fast. In the *C POSIX Library* we found the function:

```
int ffs(int i);
```

that operates on an `int` variable. The *GNU C Library* adds the following two functions that operates on arguments of possibly different size:

```
int ffsl(long int i);
int ffsll(long long int i);
```

These functions all do the same thing: starting from the least significant bit in the argument, they search for a set bit and, if found, the position is returned, otherwise they return zero.

Also the Linux kernel provides two functions that do the same thing, except that they return the most significant set bit in the argument. For our purpose, this different behaviour is peddling. The functions are:

```
int fls(int x);
int fls64(u64 x);
```

As discussed in Section 2.2, to ensure that the sequence of write operations on the top level and the bottom level bitmaps made by a CPU will be perceived by all other CPUs in the same order, a write memory barrier has to be issued. Similarly, the combiner CPU, while traversing the list, has to issue a paired read memory barrier.

Regarding the lock that protects the underlying data structure, it was decided to implement it through an `atomic_t` variable. The lock is acquired with a simple *CAS* operation, therefore with an `atomic_cmpxchg()`. Note that, as stated in Section 2.1.2, this operation issues an implicit memory barrier, so there is no way that the critical section instructions will be reordered

and positioned prior to the locking instruction. For the same reason, when we release the lock, we use an `atomic_set` operation and, after that, we issue a write memory barrier. This design allows us not to deal with `irqs` mask saving and restore, so it is a little faster than the `spinlock` solution.

As stated in the previous section, the flat combining framework introduces an asynchronous programming model. This model is unacceptable for both the *find* and the *set* operation. Regarding the former operation, we introduce a cache to always keep an updated value of the best CPU index where to migrate a task. Every time a CPU do a *set* operation, it checks the cached value and compare its deadline to decide if the cache has to be updated. If so, a *CAS* operation is immediately performed and, after that, the record is published.

Regarding the latter operation, it was decided to restrict the maximum number of records that a CPU can publish without waiting for the work to be done. In the actual implementation, this parameter can be varied changing the value of the macro `PUB_RECORD_PER_CPU`, ranging from 1 to 32.

Finally, regarding the mechanism of “combining” the *set* operations, here we can not apply such a strategy. If we compare the mean number of such operations with the number of elements in the underlying data structure (that is, the number of CPUs in the system) we can easily understand that it is not worth to sort the requests to apply that in a single pass. Anyhow, using a combiner thread that does all the work, we can benefit from keeping the cache hot in the combiner CPU, thus speeding up all the operations.

The code for this `cpudl` implementation is reported in Appendix A.2.

3.4 Fastcache

Starting from the flat combining `cpudl` implementation discussed above, we can lead some important considerations.

Most of the *find* operations are answered through the cache. In fact, we use the underlying data structure only to “reconstruct” the cache when it is invalidated. With such a design we can reach very high performance in the *find* operation. Unfortunately, the *set* operation doesn’t experiment a similar

boost: as we will see in Section 5.5, the CPU cycles needed to complete a *set* is in the same order than the skip list solution.

This drawback can be addressed using a different design that aim to use as much as possible the cache, to avoid complex algorithms that are not well suited to manage a low number of items.

A common design pattern used in parallel programming to develop scalable algorithms consists of separating the code path depending on how the concurrent requests on the data structure are interleaved. Typically we have a *fast path*, where no lock is taken, and a *slow path*, where we must take some kind of lock to ensure the correctness of the implementation. If we can ensure that the fast path will be taken most of the time, thus leads to a very fast solution.

Regarding the *set* operation on the `cpudl` data structure, recall from the discussion above that we already implicitly defined what we consider the fast path: when a CPU finds the cache in a valid state, it can compare the cached valued with its deadline value to update it, if needed. Since the update is performed through atomic operations, no lock will be taken. If the cache must not be updated, we are still in a path where no lock is needed.

A slow path must be followed when a *set* operation takes place and the cached CPU is just the same that calls the function. In this case, the value of the deadline related to that CPU must be updated, and we can not know if another CPU holds a better deadline value. So, we need to rely on a data structure where the deadline of all CPUs are stored to find which is the best one at the time. Since multiple CPUs can call the *set* operation concurrently, we have to ensure that only one CPU will be authorized to manipulate the cache, in other words, we need to protect the slow path with a lock.

For our purpose, we choose to implement the underlying structure with a simple array, to be searched with a sequential search. This choice may seem self-defeating but, as the experiments in Section 5.6 show, it is not. Such an array allows a very fast update of the deadline value associated to each CPU in the system: an `atomic_set` plus a write memory barrier is enough. This means that the *fast path* is indeed very fast. Obviously, as the number of underlying CPUs increases, the sequential search will be increasingly slower

and so will be the *slow path*. However, when the number of CPUs increases it is more likely that, between two subsequent *set* operations coming from the same CPU (the second of which would invalidate the cache), there will be another *set* operation from a different CPU that instead updates the cache. This update will change the CPU index cached value, preventing the subsequent *set* operation from invalidating the cache. In this manner the slow path will be taken in very few cases.

To guarantee the consistency of the *cpudl* data structure, we have to ensure that:

- as soon as a CPU enters the `cpudl_set` function, it has to update its deadline value stored in the array with an `atomic_set`;
- when more than one CPU concurrently executes the `cpudl_set` while the cache is invalidated, we first try to acquire the lock to refill it, but, if the lock is taken, we simply retry until the cache is valid. This way, we have a chance to “fast-update” the cache with our new deadline value through a simple *CAS*.

Finally, another improvement can be made to speed up the *slow path*. Suppose that the number of per-CPU tasks is low: this condition leads to a higher number of runqueues with only one deadline tasks enqueued in it. So, we would have an increasing rate of *set* operations with the `is_valid` flag set to zero, thus leading to a higher rate of cache invalidations. So, to obtain good performance even in such a situation, we used the CPU bitmask also for the pull operation: while scanning the deadlines array through the *slow path*, that bitmask tell us which CPU has no `next` deadline tasks. Doing this, we don't need to scan every single element of the array: we can simply skip those CPUs.

This solution has been named *fastcache*, from the words “*fast path*” and “*cache*”. The *fastcache* code is reported in Appendix A.3.

3.5 Improved pull algorithm

As discussed in Section 1.5.7, the current implementation of `SCHED_DEADLINE` lacks a data structure to speed up the *pull* operation. So, a scheduler instance that wants to migrate a task through a *pull* operation needs to sequentially search all the runqueues in the system to find the eligible tasks to pull. This is a major drawback, for two main reasons:

- With the number of CPUs increasing, an unacceptable latency will affect every pull operation;
- as seen in Section 1.2.9, the pull operation continues to pull tasks until a suitable one could be found. Even if the CPUs are clustered into root domains, this strategy can lead to a lot of useless task migrations, since only a single task will be the running one: the others will remain enqueued with little chance to execute. These tasks will be eligible for the subsequent push operations, leading to the *task bouncing* phenomenon.

Thus we can conclude that this algorithm puts a non negligible overhead on the scheduler. Theus, we decided to tackle the same approach followed for *push* operation: similar data structure has been implemented, with three key differences:

- the tasks that we have to consider when executing a *pull* operation are the second ones enqueued in each runqueue;
- tasks are sorted in increasing deadline order;
- since we are searching for a task to pull in the current runqueue, and we have no pointer to such a task, we can not check, inside `cpudl` data structure, the task affinity, as we do for the *push* operation.

An example of such a `cpudl` implementation can be seen in Figure 3.2 on the following page where we consider a 4-CPU system.

All the data structures presented in the previous sections have been developed with a hook to a deadline compare function: this way we can use the same code for both push and pull operations.

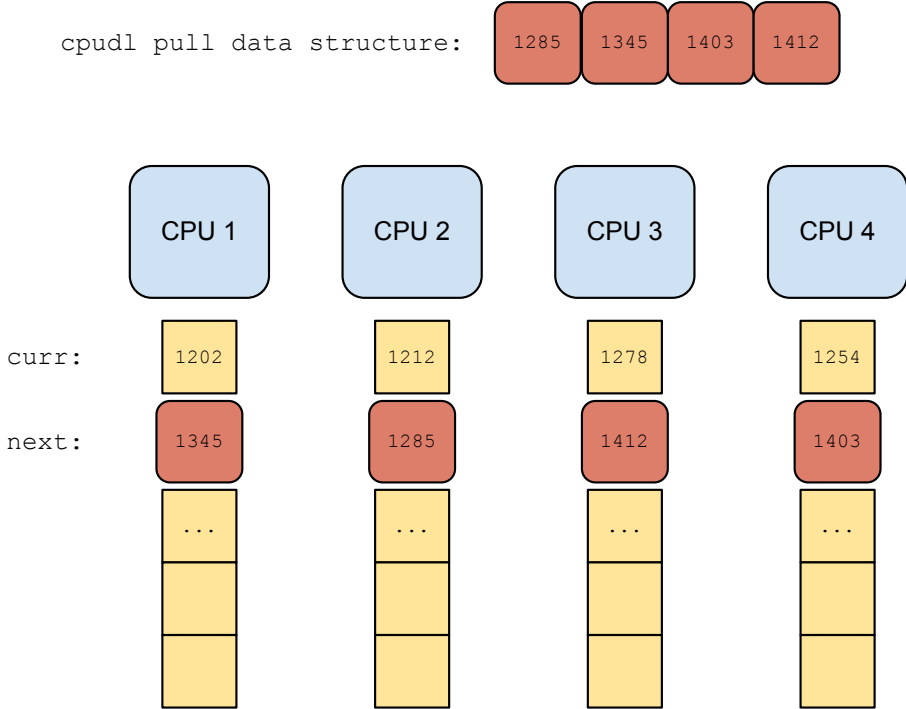


Figure 3.2: cpudl structure for pull operation.

The related source code is reported in Appendix A.4.

Chapter 4

PRACTISE framework

In this chapter we will describe PRACTISE, a novel framework to help developing new scheduling algorithm for the Linux kernel in user space. We briefly present a survey about the state-of-art kernel development tools, highlighting the major advantages and drawbacks of each one. Then, we will show why a PRACTISE may be useful and how it is designed.

Finally, we will compare the results of some experiments made both in PRACTISE and in the Linux kernel.

4.1 Tools for Linux kernel development

Scheduling on multi-core and multiprocessor system is an open research field both from the point of view of the theory and for the technical difficulties in implementing an efficient scheduling algorithm in the kernel.

Regarding the second problem, we're going to point out the difficulties that kernel developers encounter in their task.

The scheduler is a fundamental part of the operating system kernel: a buggy scheduler will soon crash the system, usually at random and unexpected points. The major difficulty that a prospective developer encounters when developing a new scheduling algorithms derives from the fact that, when the system crashes, it is difficult to reconstruct the sequence of events and states that led to the crash.

The developer has to carefully analyse system logs and traces, but this task is far from simple due to the complexity of the kernel itself: the number of functions that compose a commercial *OS* like Linux is huge. More importantly, it is often impossible to impose a precise sequence of events to deterministically reproduce a particular status. Hence, it is practically impossible to run a sequence of test-cases.

This problem is exacerbated in multi-core architectures where the scheduler service routines run in parallel on the different processors, and make use of shared data structures that are accessed in parallel. In these cases, it is necessary to ensure that the data structures remain consistent under every possible interleaving of the service functions: as we will see in the following sections, this problem is far from trivial.

Now let us present a quick list of the most widely adopted solutions for Linux kernel development, with particular reference to the tools specifically designed for the developing of a new scheduling algorithm.

4.1.1 LinSched

LinSched was originally developed by the Real Time System Group at University of North Carolina at Chapel Hill, and it's currently maintained by P. Turner from Google. This tool lets developers modify the behaviour of the Linux scheduler and test changes in user-space. One of the major strength points of this tool is that it introduces very few modifications in the kernel sources: the developer can write kernel code and, once satisfied by tests, he has kernel ready patches at hand. One key point of *LinSched* is that it runs as a single thread user-space program. This leads to a facilitated debugging process, because we can effectively use user-space common tool like, among the others: *GDB*, *gprof* and *Valgrind*.

On the other hand, single-threading is a notable drawback when we are focusing on the analysis of behaviour assumining a high degree of concurrency. *LinSched* can indeed verify locking, but it cannot precisely model multi-core contention.

4.1.2 LITMUS^{RT}

We have already described LITMUS in Section 1.3.6, here we are going to point out the facilities that come with LITMUS to facilitate the development of a new real-time scheduling algorithm.

LITMUS provides an integrated tracing infrastructure (named *Feather-Trace*) with which performance and overhead data can be collected for off-line processing.

Being a research tool rather than a production-quality system, LITMUS does not target Linux mainline inclusion nor POSIX-compliance: in other words code patches created with it cannot be seamlessly applied to a “Vanilla” Linux kernel.

4.1.3 KVM + GDB

The very first step after having modified the kernel is usually to run it on a virtualized environment. This solution allows to create a virtual machine with suitable characteristics for the developed code (like a high number of virtual cores to simulate a high concurrency platform) and with a faster booting process compared to that of a physical machine.

In addition to this, KVM has an option to expose a server on a port where GDB can connect to control the kernel execution. Even if this solution has some limitations, like the impossibility of using software breakpoints, it is indeed an invaluable help in the debugging process.

Unfortunately, this solution can hardly be used in a presence of high concurrency, moreover, it can occasionally affect the repeatability of certain bugs.

4.2 PRACTISE architecture

PRACTISE emulates the behaviour of the Linux scheduler subsystem on a multi-core architecture with M parallel cores. The tool can be executed on a machine with N cores, with N that can be less, equal or greater than M . The tool can be executed in one of the following modes:

- *testing*
- *performance analysis*

Each processor in the simulated system is modelled by a software thread that performs a cycle in which:

- scheduling events are generated at random
- the corresponding scheduling functions are invoked
- statistics are collected

In *testing mode*, a special “testing” thread is executed periodically and it performs consistency checks on the shared data structures. In the *performance analysis* mode, instead, each thread is *pinned* on a processor, and the memory is locked to avoid spurious page faults; for this reason, to obtain realistic performances it is necessary to set $M \leq N$.

4.2.1 Ready queues

In the current version of PRACTISE the structure of distributed queues as it is in the kernel has been maintained. The same *push* and *pull* algorithms used in Linux to migrate tasks between runqueues, as described in Section 1.2.3, have been implemented too. To speed up the *push* operation we have seen that the current release of SCHED_DEADLINE uses a max heap to store the deadlines of the tasks executing on the processors. In a similar manner, the current release of SCHED_RT scheduling class uses a priority map¹ to record, for each processor, the priority value of the highest priority tasks. We find those global data structure even in PRACTISE, with one key difference: in PRACTISE we developed and tested a *cpudl* data structure to speed up also the *pull* operations in SCHED_DEADLINE scheduling class. This solution and its potential advantages has been already described in Section 3.5.

During the simulation, tasks are inserted into (removed from) the ready queues using the `enqueue ()` (`dequeue ()`) function, respectively. In Linux,

¹implemented in `kernel/sched/cpupri.c`

the queues are implemented as red-black trees. In PRACTISE, for the sake of simplicity, we have implemented them as priority heaps, using the data structure proposed by B. Brandenburg². Since we are mainly interested in observing the migration tasks pattern of activity, this difference don't affect our analysis.

In the following subsections, where we're going to analyze in great detail the tool internals, we will refer to the global data structures used to speed up the push and pull operations as `push_struct` and `pull_struct`, respectively.

4.2.2 Locking and synchronization

PRACTISE uses a range of locking and synchronization mechanisms that mimic the corresponding mechanisms in the Linux kernel. An exhaustive list is given in Table 4.1. These differences are major culprits for the slight changes needed to port code developed on the tool in the kernel, as we will see in Section 4.4.1.

It has to be noted that `wmb` and `rmb` kernel memory barriers have no corresponding operations in user-space; therefore we have to issue a full memory barrier (`--sync_synchronize`) for every occurrence of them.

4.2.3 Event generation and processing

PRACTISE cannot execute or simulate a real application. Instead, each threads (that emulates a processor) periodically generates random scheduling events according to a certain distribution, and calls the scheduler functions. The goals of PRACTISE are to make sure that the developer can easily debug, test, compare and evaluate real-time scheduling algorithms for multi-core processors. Therefore, we identified two main events: *activation* and *blocking*.

When a task is activated, it must be inserted in one of the kernel queues; since such an event can cause a preemption, the scheduler is invoked, data structures are updated, etc. Something similar happens when a task self-

²Code available here: <http://bit.ly/IozLxM>.

suspends (for example because it blocks on a semaphore, or it suspends on a timer).

The pseudo-code for the task activation is represented in Listing 4.1.

Listing 4.1: Task activation pseudo-code

```
on_activation(task) {
    enqueue(task, local_queue);
    pull(); /* pre-schedule */
    push(); /* post-schedule*/
}
```

The code mimics the sequence of events that are performed in the Linux code:

- First, the task is inserted in the local queue
- Then, the scheduler performs a *pre-schedule*, corresponding to `pull()`, which looks at the global data structure `pull_struct` to find the task to be pulled; if it finds it, does a sequence of `dequeue()` and `enqueue()`.
- Then, the Linux scheduler performs the real schedule function; this corresponds to setting the `curr` pointer to the executing task. In PRACTISE this step is skipped, as there is no real context switch to be performed.
- Finally, a *post-schedule* is performed, consisting of a `push()` operation, which looks at the global data structure `push_struct` to see if some task need to be migrated, and in case the response is positive, performs a `dequeue()` followed by an `enqueue()`. A similar thing happens when a task blocks (see the pseudo-code for the task blocking operation in Listing 4.2).

Listing 4.2: Task blocking pseudo-code

```
on_block(task) {
```

```

    dequeue(&task, local_queue);
    pull(); /* pre-schedule */
    push(); /* post-schedule*/
}

```

Linux	PRACTISE	Action
raw_spin_lock	pthread_spin_lock	lock a structure
raw_spin_unlock	pthread_spin_unlock	unlock a structure
atomic_inc	__sync_fetch_and_add	add a value in memory atomically
atomic_dec	__sync_fetch_and_sub	subtract a value in memory atomically
atomic_read	simple read	read a value from memory
wmb	__sync_synchronize	issue a memory barrier
rmb	__sync_synchronize	issue a read memory barrier
mb	__sync_synchronize	issue a full memory barrier

Table 4.1: Locking and synchronisation mechanisms (Linux vs. PRACTISE).

As anticipated, every processor is simulated by a periodic thread. The thread period can be set varying a constant in the `parameters.h` header file and represents the average frequency of events arriving at the processor. At every cycle, the thread randomly select one between the following events:

- `activation`
- `early_finish`
- `idle`

In the first case, a task is generated with a random value of the deadline and function `on_activation()` is called. In the second case, the task currently executing on the processor blocks: therefore function `on_block()` is called. In the last case, nothing happens. Additionally, in all cases, the deadline of the executing task is checked against the current time: if the deadline has passed, then the current task is blocked, and here again, function `on_block()` is called.

With PRACTISE, it is possible to specify the period of the thread cycle, the probability of an activation event, and the probability of an early finish.

4.2.4 Data structures in PRACTISE

PRACTISE has a modular structure, tailored to provide flexibility in developing new algorithms. The interface exposed to the user consists of hooks to function that each global structure must provide. The most important hooks are:

data_init initialises the structure, e.g. spinlock init, dynamic memory allocation, etc.

data_cleanup performs clean up tasks at the end of a simulation.

data_preempt called each time a local queue changes its status (e.g. an arriving task has higher priority than the currently executing one, so it causes a preemption); this function modifies the global structure to reflect new local queue status.

data_find used by a scheduling policy to find the best CPU to (from) which push (pull) a task.

data_check implements the *checker* mechanism (described below).

One of the major features provided by PRACTISE is the *checking* infrastructure. Since each data structure has to obey different rules to preserve consistency among successive updates, the user has to equip the implemented algorithm with a proper checking function. When the tool is used in testing mode, the *data_check* function is called at regular intervals. Therefore, an on-line validation is performed in presence of real concurrency thus increasing the probability of discovering bugs at an early stage of the development process. User-space debugging techniques can then be used to fix design or developing flaws.

To give an example, the *checking* function for SCHED_DEADLINE *cpudl* structure ensures the max-heap property: if *B* is a child node of *A*, then

$deadline(A) \geq deadline(B)$; it also check consistency between the heap and the array used to perform updates on intermediates nodes (see [19] for further details). We also implemented a checking function for *cpupri* data structure: periodically, all ready queues are locked, and the content of the data structure is compared against the corresponding highest priority task in each queue, and the consistency of the flag `overloaded` in the `struct root_domain` is checked. We found that the data structure id always perfectly consistent to an external observer.

4.3 Performance analysis with PRACTISE

To collect the measurements we use the TSC³ of IA-32 and IA-64 Instruction Set Architectures. The TSC is a special 64-bit per-CPU register that is incremented every clock cycle. This register can be read with two different instructions: `RDTSC` and `RDTSCP`. The latter reads the TSC and other information about the CPU that issues the instruction itself. However, there a number of possible issues that needs to be addressed in order to have a reliable measure:

- *CPU frequency scaling and power management.* Modern CPUs can dynamically vary frequency to reduce energy consumption. Recently, CPUs manufacturer has introduced a special version of TSC inside their CPUs: *constant TSC*. This kind of register is always incremented at CPU maximum frequency, regardless of CPU actual frequency. Every CPU that supports that feature has the flag *constant_tsc* in `/proc/cpuinfo` `proc` file of Linux. Unfortunately, even if the update rate of TSC is constant in these conditions, the CPU frequency scaling can heavily alter measurements by slowing down the code unpredictably; hence, we have conducted every experiment with all CPUs at fixed maximum frequency and no power-saving features enabled. To do this the `cpufreq` utility has been used.

³Time Stamp Counter

- *TSC synchronisation between different cores.* Since every core has its own TSC, it is possible that a misalignment between different TSCs may occur. Even if the kernel runs a synchronisation routine at start up (as we can see in the kernel log message), the synchronisation accuracy is typically in the range of several hundred clock cycles. To avoid this problem, we have set CPU affinity of every thread with a specific CPU index. In other words we have a 1:1 association between threads and CPUs, fixed for the entire simulation time. In this way we also prevent thread migration during an operation, which may introduce unexpected delays.
- *CPU instruction reordering.* To avoid instruction reordering, we use two instructions that guarantees serialisation: RDTSCP and CPUID. The latter guarantees that no instructions can be moved over or beyond it, but has a non-negligible and variable calling overhead. The former, in contrast, only guarantees that no previous instructions will be moved over. In conclusion, as suggested in [25], we used the following sequence to measure a given code snippet:

```

CPUID
RDTSC
code
RDTSCP
CPUID

```

- *Compiler instruction reordering.* Even the compiler can reorder instructions; so we marked the inline asm code that reads and saves the TSC current value with the keyword *volatile*.
- *Page faults.* To avoid page fault time accounting we locked every page of the process in memory with a call to `mlockall`.

PRACTISE collects every measurement sample in a global multidimensional array, where we keep samples coming from different CPUs separated. After all simulation cycles are terminated, all the samples are written to an output file.

By default, PRACTISE measures the following statistics:

- duration and number of *push* and *pull* operations;
- number of *enqueue* and *dequeue* operations;
- duration and number of *data_preempt*, *data_finish* and *data_find* operations

It is possible to add different measures in the code of a specific algorithm by using PRACTISE's macros.

For example, suppose that we want to measure the number of clock cycles that a code snippet takes to be executed: we refer to this piece of code as *operation_under_measure* in the subsequent explanation.

To enable the measure samples collection we have to:

- insert the following lines of code in *include/measure.h* header file:

```
EXTERN_MEASURE_VARIABLE(operation_under_measure)
EXTERN_DECL(ALL_COUNTER(operation_under_measure))
```

and also the following lines in *src/measure.c* source file:

```
MEASURE_VARIABLE(operation_under_measure)
ALL_COUNTER(operation_under_measure)
```

to declare the array that will holds all the measurement samples and the variable that will holds the number of measurement samples collected (useful for average calculation).

- insert the following lines of code at the beginning and at the end of the main function in *src/practise.c* source file, respectively:

```
MEASURE_ALLOC_VARIABLE(operation_under_measure)

MEASURE_FREE_VARIABLE(operation_under_measure)
```

to allocate memory for the measurement samples global array.

- surround the code snippet to measure with the following two instructions:

```

MEASURE_START(operation_under_measure, CPU-index)

<code_to_measure>

MEASURE_END(operation_under_measure, CPU-index)

```

where `CPU-index` is the CPU that is executing the code under measure. This is useful if we want to analyze how much operations a specific CPU carries on.

- finally, add those lines of code in `src/practise.c`:

```

MEASURE_STREAM_OPEN(operation_to_measure, online_cpus);
for(i = 0; i < online_cpus; i++){
    MEASURE_PRINT(out_operation_to_measure, operation_to_measure, i);
    fprintf(out_operation_to_measure, "\n");
}
MEASURE_STREAM_CLOSE(operation_to_measure);

```

to print all the measurements sample in a properly formatted way.

4.4 Evaluation

In this section, we show how difficult is to port a scheduler developed with the help of PRACTISE into the Linux kernel; then, we report performance analysis figures and discuss the different results obtained in user space with PRACTISE and inside the kernel.

4.4.1 Porting to Linux

The effort in porting an algorithm developed with PRACTISE in Linux can be estimated by counting the number of different lines of code in the two implementations. We have two global data structures implemented both in PRACTISE and in the Linux kernel: `cpudl` and `cpupri`.

We used the `diff` utility to compare differences between user-space and kernel code of each data structure. Results are summarised in Table 4.2.

Less than 10% of changes were required to port `cpudl` to Linux, these differences mainly due to the framework interface (that is, pointers conversion).

Structure	Modifications	Ratio
<i>cpudl</i>	12+ 14-	8.2%
<i>cpupri</i>	17+ 21-	14%

Table 4.2: Differences between user-space and kernel code.

```

[...]
-void cpupri_set(void *s, int cpu, int newpri)
+void cpupri_set(struct cpupri *cp, int cpu,
+               int newpri)
{
- struct cpupri *cp = (struct cpupri*) s;
  int *currpri = &cp->cpu_to_pri[cpu];
  int oldpri = *currpri;
  int do_mb = 0;
@@ -63,57 +61,55 @@
  if (newpri == oldpri)
      return;

- if (newpri != CPUPRI_INVALID) {
+ if (likely(newpri != CPUPRI_INVALID)) {
    struct cpupri_vec *vec =
        &cp->pri_to_cpu[newpri];

    cpumask_set_cpu(cpu, vec->mask);
- __sync_fetch_and_add(&vec->count, 1);
+ smp_mb__before_atomic_inc();
+ atomic_inc(&(vec)->count);
    do_mb = 1;
  }
[...]
```

Figure 4.1: Comparison using `diff`.

Slightly higher changes ratio for *cpupri*, due to the quite heavy use of atomic operations. An example of such changes is given in Figure 4.1 (lines with a `-` correspond to user-space code, while those with a `+` to kernel code).

The difference on the synchronisation code can be reduced by using appropriate macros. For example, we could introduce a macro that translates to `__sync_fetch_and_add` when compiled inside PRACTISE, and to the corresponding Linux code otherwise. However, we decide to maintain the different code to highlight the differences between the two frameworks, rather than hide them.

In conclusion, the amount of work shouldered on the developer to transfer

the implemented algorithm to the kernel, after testing, is quite low reducing the probability of introducing bugs during the porting.

Chapter 5

Experimental Results

In this chapter, we will summarize the results of various experiments conducted with all the solutions presented in Chapter 3. First we will compare the results of some experiments made both in PRACTISE and in the Linux kernel. Those experiments show the ability of PRACTISE to predict the relative performance of different algorithms.

Then, we will focus on the scalability of the algorithms presented in Chapter 3: as the number of the underlying CPUs increases, we will see that the tasks migration mechanism still shows good performance for both *push* and *pull* operations.

Before implementing the algorithms in kernel space, extensive tests with PRACTISE were conducted. We used the tool to correct all the bugs found by the checking subsystem (discussed in Section 4.2.4). Then, we conducted a performance analysis to understand which solution performs best in the user space simulation. As we will show in this chapter, each algorithm provides interesting results that allowed us to reach, step by step, a solution that scales well in every situation. So, it has been decided to port all the algorithms in kernel space, to run a performance test with each of them. Those tests have confirmed once again the ability of PRACTISE to predict the relative performance of the code tested with it.

While porting the algorithms in kernel space, the code developed with PRACTISE has undergone very little changes, almost all related to the dif-

ferences presented in Table 4.1 in Section 4.2.3. In addition to those, a single modification has to be highlighted: in kernel space we do not have the `rand` function of the *C standard library*, so the `kernel_current_time` function has been used to obtain a pseudo-random value to generate skip list items. For our purpose this function is sufficient, because we do not need a strong random generator.

5.1 Experiments with PRACTISE

The aim of the experimental evaluation is to compare performance measures obtained with PRACTISE with what can be extracted from the execution on a real machine.

Of course, we cannot expect the measures obtained with PRACTISE to compare directly with the measure obtained within the kernel; there are too many differences between the two execution environments to make the comparison possible. For example, we can consider the completely different synchronization mechanisms or the unpredictability of hardware interrupts that the kernel has to manage. However, comparing the performance of two alternative algorithms within PRACTISE can give us an idea of their relative performance within the kernel.

In Linux we run experiments on a Dell PowerEdge R815 server equipped with 64GB of RAM, and 4 AMD^R OpteronTM 6168 12-core processors running at 1.9 GHz, for a total of 48 cores. We generated 20 random task sets (using the `randfixedsum` [12] algorithm) with periods log-uniform distributed in [10ms, 100ms], per CPU utilisation of 0.6, 0.7 and 0.8 and considering 2, 4, 8, 16, 24, 32, 40 and 48 processors. Then, we ran each task set for 10 seconds using a synthetic benchmark ¹ that lets each task execute for its WCET every period. We varied the number of active CPUs using the Linux CPU hot plug feature and we collected scheduler statistics through the `sched_debug` proc file.

The results for the Linux kernel are reported in Figures 5.1 and 5.2, for

¹rt-app: <https://github.com/gbagnoli/rt-app>

modifying and querying the data structures, respectively.

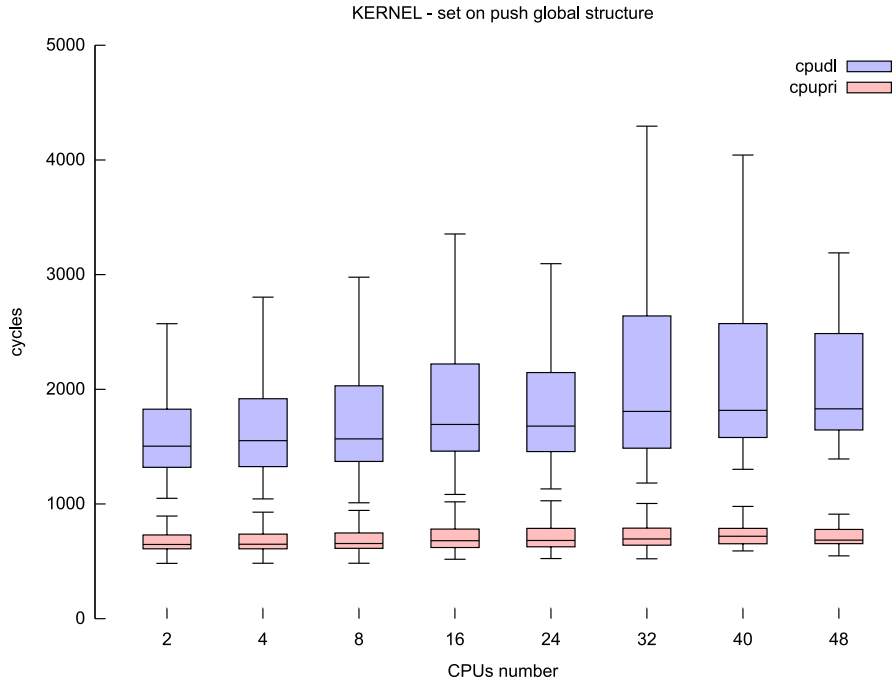


Figure 5.1: Global data structure modify

The figures show the number of cycles (y axis) measured for different number of processors ranging from 2 to 48 (x axis). The measures are shown in boxplot format: a box indicates all data comprised between the 25% and the 75% percentiles, whereas an horizontal lines indicates the median value; also, the vertical lines extend from the minimum to the maximum value.

In PRACTISE we run the same experiments. As depicted in Section 4.2.3 random scheduling events generation is instead part of PRACTISE. We varied the number of active processors from 2 to 48 as in the former case.

We set the following parameters: 10 milliseconds of thread cycle; 20% probability of new arrival; 10% probability of finish earlier than deadline (for *cpudl* data structure) or runtime (for *cpupri* data structure); 70% probability of doing nothing. These probability values lead to rates of about 20 task activation / (core * s), and 20 task blocking / (core * s).

The results are shown in Figures 5.3 and 5.5 for modifying the *cpupri* and *cpudl* data structures, respectively; and in Figures 5.4 and 5.6 for querying

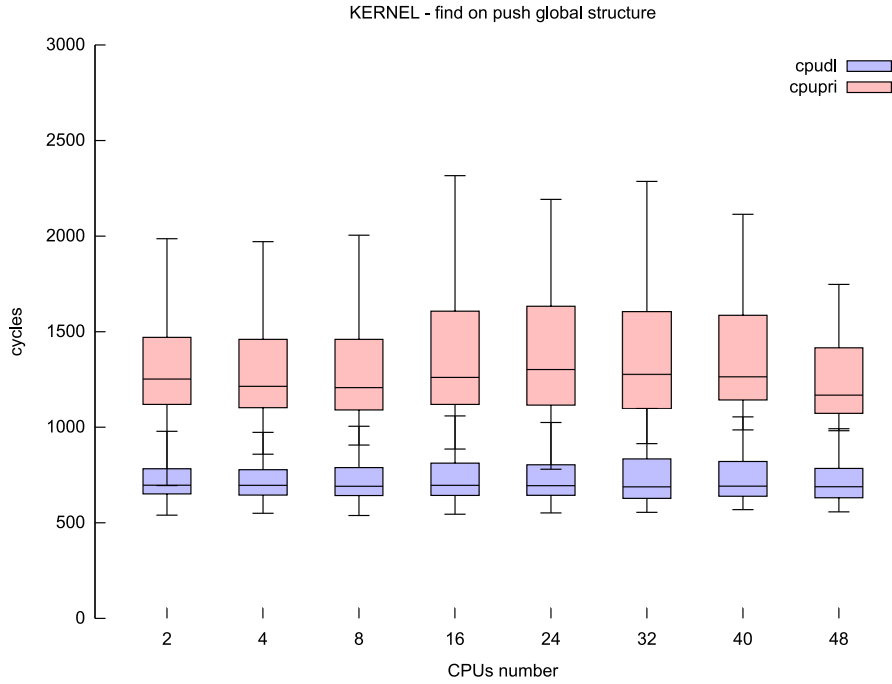


Figure 5.2: Global data structure query

the *cpupri* and *cpudl* data structures, respectively.

Insightful observations can be made comparing performance figures for the same operation obtained from the kernel and from simulations. Looking at Figure 5.1 we see that modifying the *cpupri* data structure is generally faster than modifying *cpudl* data structures: every measure corresponding to the former structure falls below 1000 cycles while the same operation on *cpudl* takes about 2000 cycles. Same trend can be noticed in Figures 5.3 and 5.5. Points dispersion is generally a bit higher than in the previous cases; however median values for *cpupri* are strictly below 2000 cycles while *cpudl* never goes under that threshold. We can see that PRACTISE overestimates this measures: in Figure 5.3 we see that the estimation for the *set* operation on *cpupri* are about twice the ones measured in the kernel; however, the same happens for *cpudl* (in Figure 5.5); therefore, the relative performance of both does not change.

Regarding query operations the ability of PRACTISE to provide an estimation of actual trends is even more evident. Figure 5.6 shows that a *find*

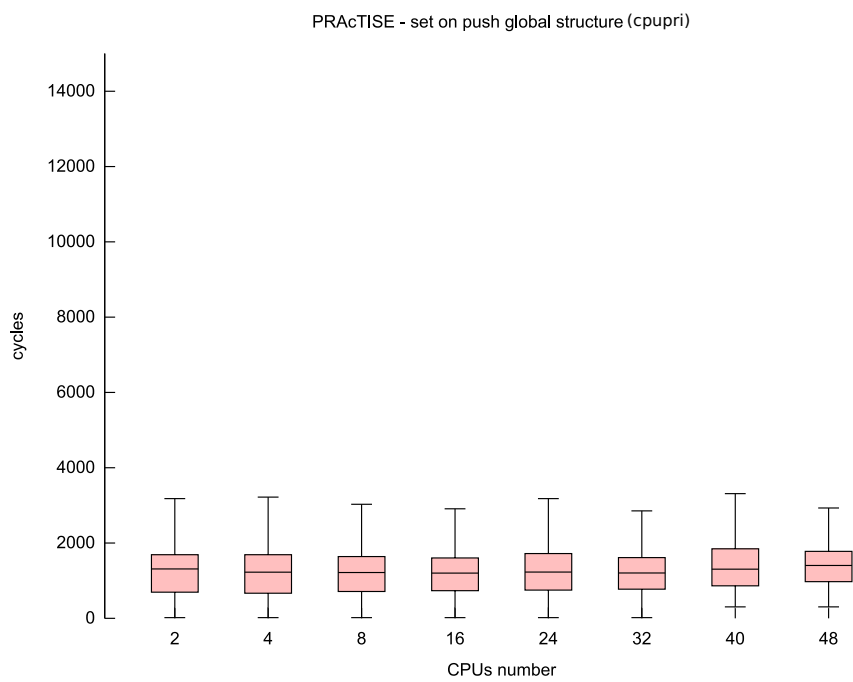


Figure 5.3: Global data structure *cpupri* modify

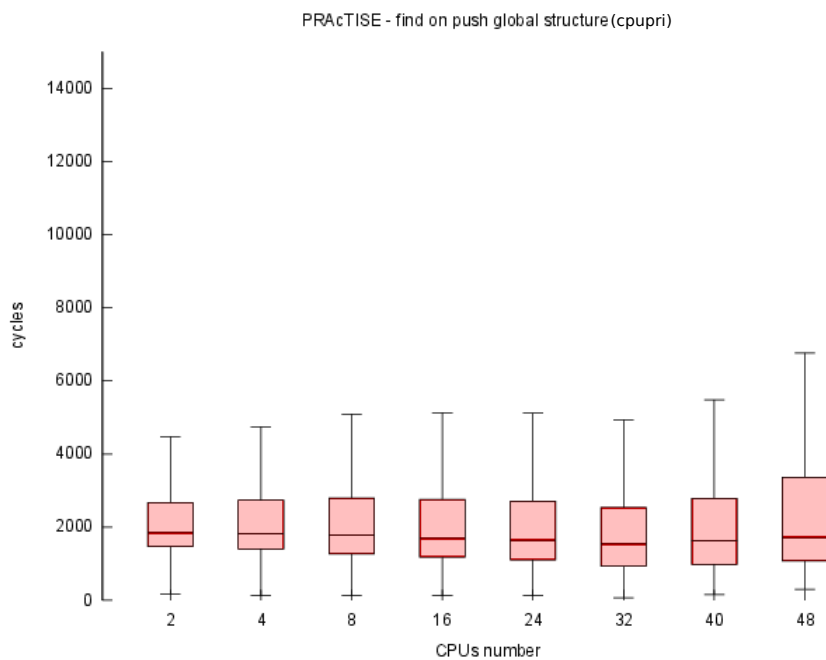


Figure 5.4: Global data structure *cpupri* query

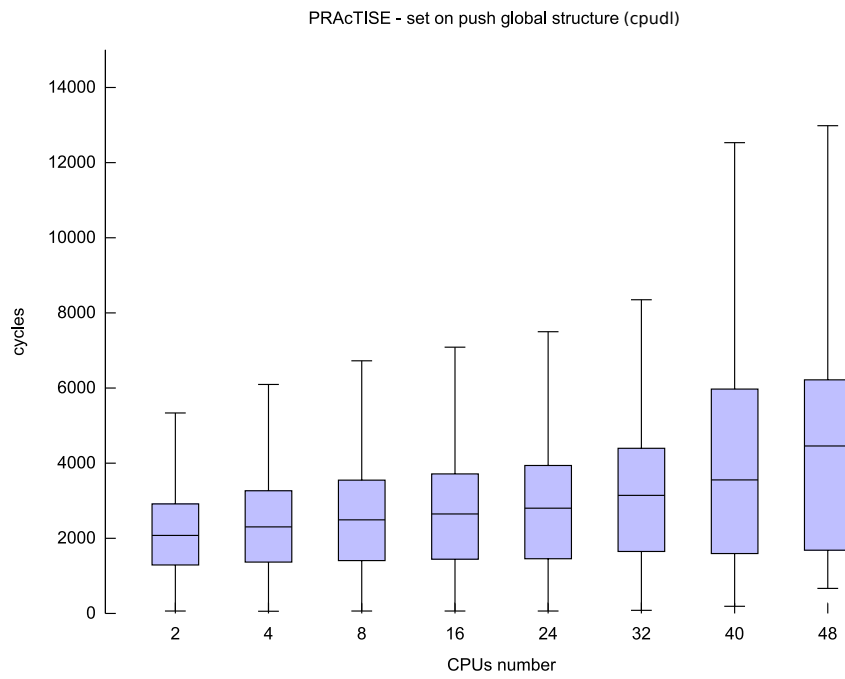


Figure 5.5: Global data structure *cpudl* modify

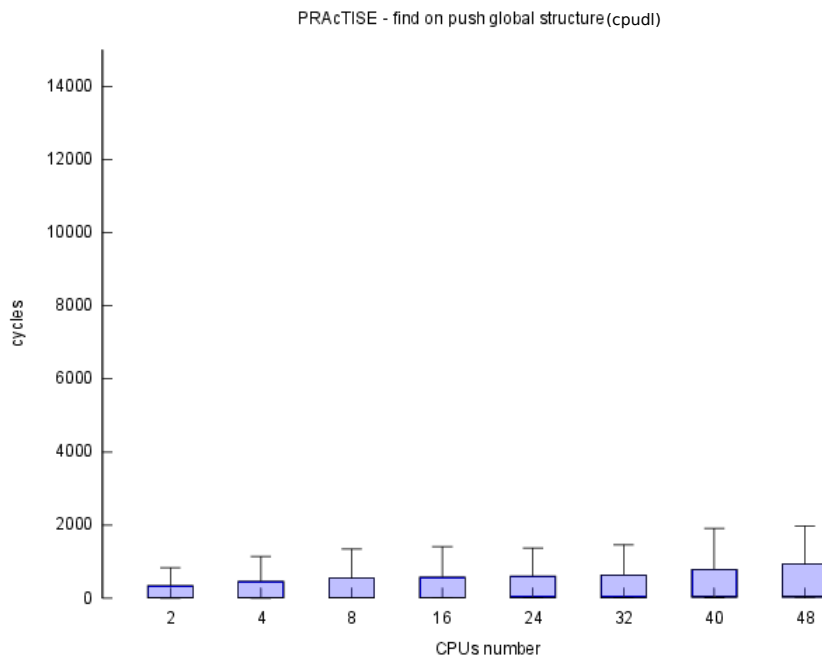


Figure 5.6: Global data structure *cpudl* query

on *cpudl* is generally more efficient than the same operation on *cpupri*; this was expected, because the former simply reads the top element of the heap. Comparing Figure 5.4 with Figure 5.6 we can state that latter operations are the most efficient also in the simulated environment. As a concluding use-case, it is worth mentioning that PRACTISE has already been used as a testing environment for the last SCHED_DEADLINE release on the LKLM². The *cpudl* global data structure underwent major changes that needed to be verified. The tested code has been finally merged within the patch set.

5.2 Kernel Experiments

Regarding the kernel experiments, since the results for the different values of CPU utilization are very similar, in the subsequent sections we are going to show only the graphs related to task sets with a U value of 0.8.

We will focus on the graphs related to the performance of the *cpudl* data structures, that is: the CPU cycles of the *find* operation and the *set* operation. We will show the number of *push* and *pull* operations and we will point out the benefit of using a *cpudl* data structure to speed up the pull operations.

5.3 Comparison between max-heap and skip list

In this section we are going to focus on the *push* operation: we will compare the *cpudl* max-heap with the skip list one.

The results are shown in Figure 5.7.

We can see that the *find* operation is always faster in the skip list implementation: the median value is always under 600 CPU cycles, while the max-heap never goes under that threshold. Both implementations are not

²LKLM (Linux Kernel Mailing List) thread available at: <https://lkml.org/lkml/2012/4/6/39>

affected by the increase in the CPUs number: we can see that the results are the same from 2 to 48 CPUs. This shows that they are both scalable.

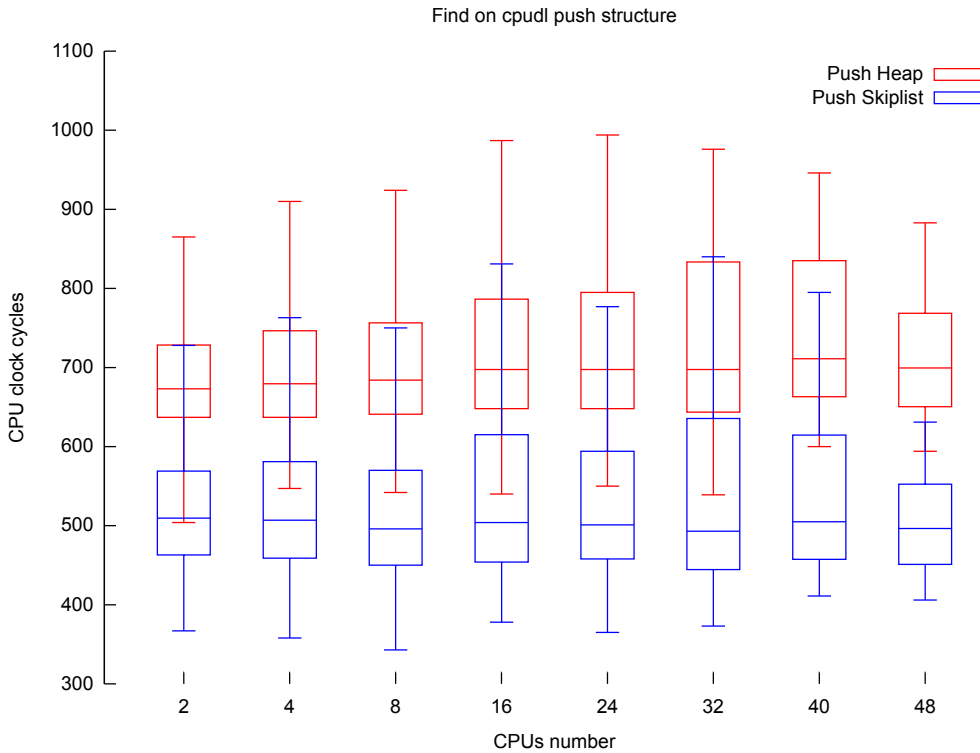


Figure 5.7: *set* operation on max-heap and skip list kernel

Regarding the *set* operation, the result is reversed (Figure 5.8): we see that the max-heap is very fast and never exceeds the 2000 cycles threshold. We can see that also the scalability of this solution is good: as the number of CPUs increases, the heap still performs quite well, even if a slight worsening can be noted when the CPUs are 24 or more.

The skip list implementation is not as fast as the heap in updating the structure: the number of CPU cycles needed to perform the same operations are about double. Regarding the scalability, we can see that the operation tends to slightly slow while the CPUs are more than 16, but still maintains a good performance even with 48 CPUs.

To perform a fair comparison between the two implementations, we need to know the number of operations carried out on the data structure. In

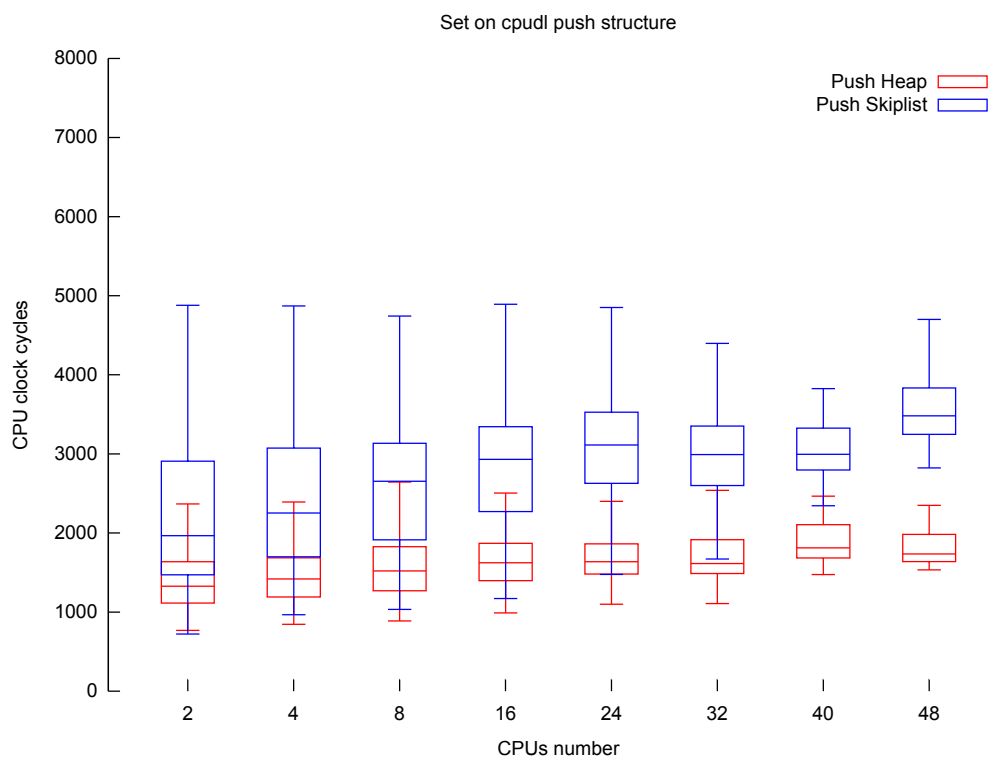


Figure 5.8: *find* operation on max-heap and skip list kernel

Figure 5.9 and Figure 5.10 we can see the number of *set* and *find* per CPU operations, respectively. Since the number of *set* operations greatly exceeds the *find* one, and since the spread between max-heap and skip list performance is much wider in the *set* case than the *find* one, we can definitely state that the heap is a better solution for the *push* operation.

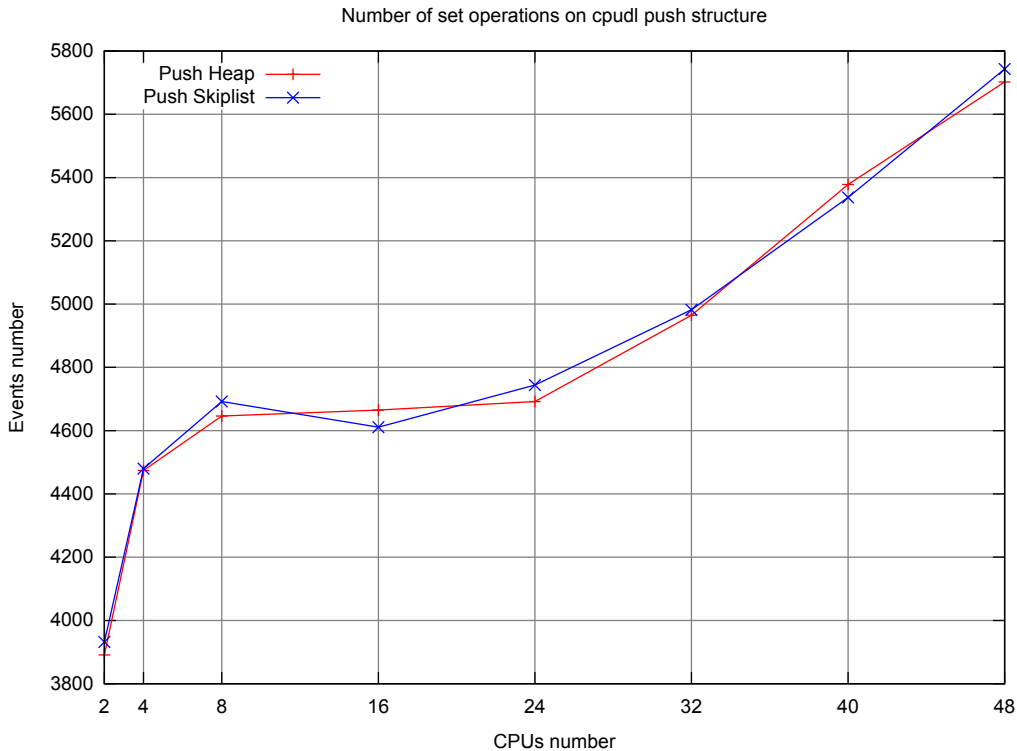
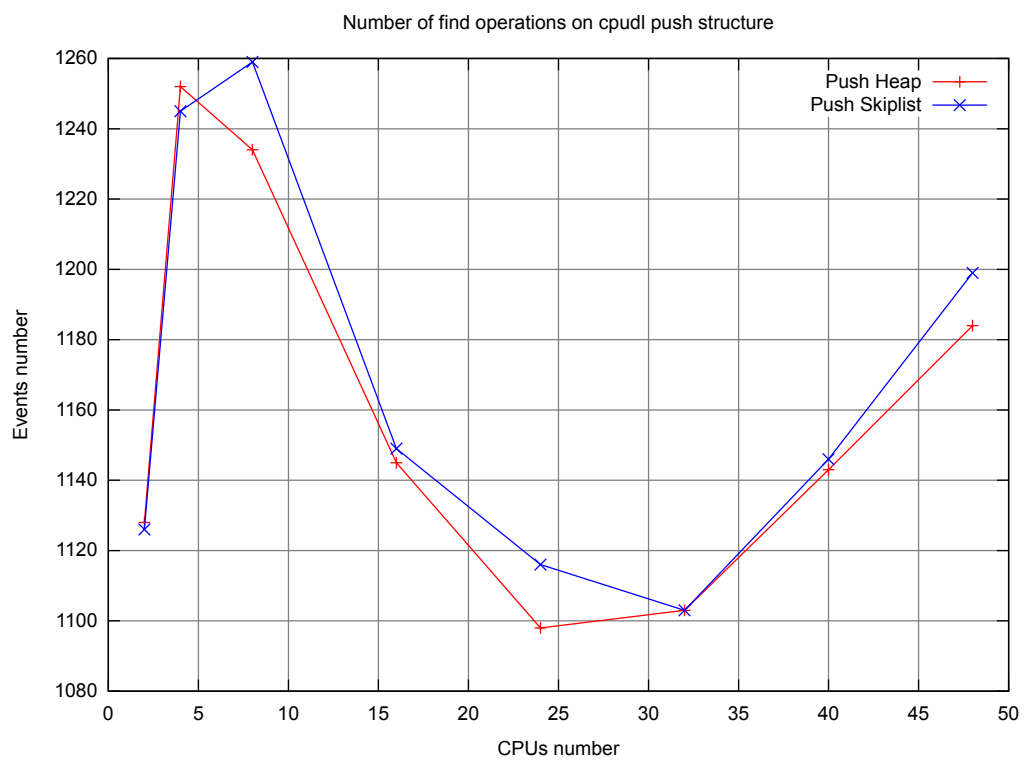


Figure 5.9: *set* operations number

5.4 Improved Pull algorithm performance

As we have seen in Section 1.5.6 and in Section 3.5, the current implementation of `SCHED_DEADLINE` lacks a data structure to speed up the pull operation. So, we decided to address this problem following the same approach developed for the push operation. We chose the skip list implementation of the `cpudl` data structure and, with a kernel modified as such, we conduct the same experiments described in Section 5.1.

Figure 5.10: *find* operations number

The results are shown in Figure 5.11 and in Figure 5.12 where we can see the number of succesfull per-CPU task migrations due to push and pull operations, respectively. Regarding the push-related migrations, we see that there is no difference; on the other hand, since we used a `cpudl` data structure also for *pull* operation, there is no need to explore all runqueues in the system: so, the number of migrations is lower for every number of online CPUs.

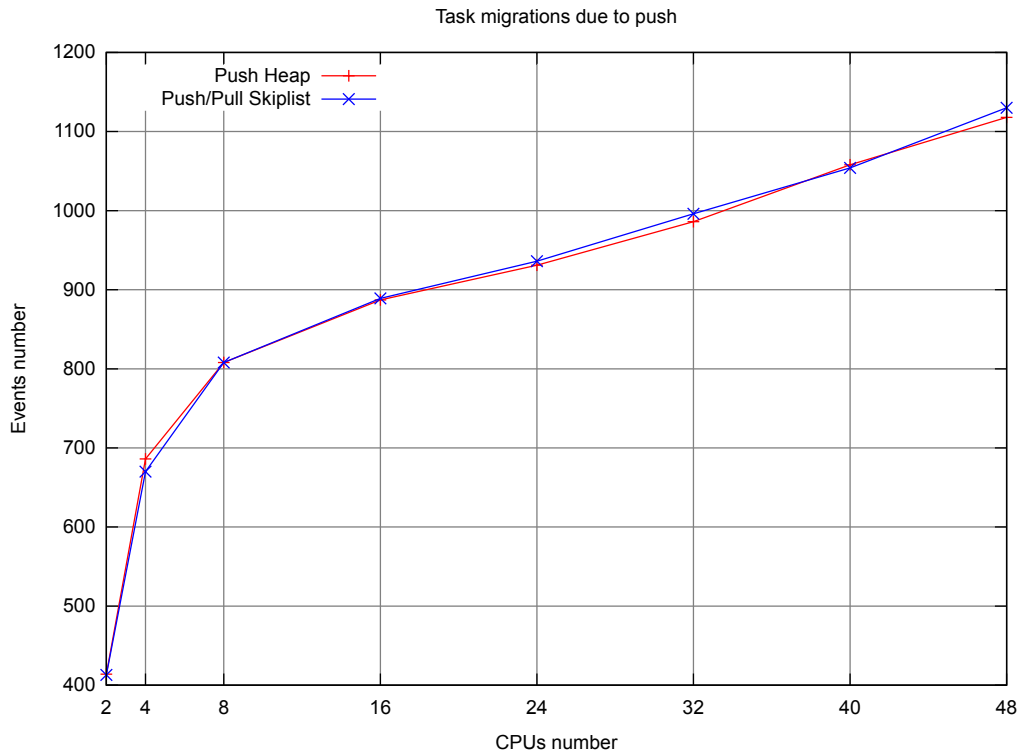


Figure 5.11: Number of task migrations due to push operation

5.5 Bitmap flat combining performance

In this section we discuss the performance of the bitmap flat combining solutions. This implementation is the basis for the fastcache algorithm.

In Figures 5.13 and 5.14 we can observe the performance related to the push and the pull operations, respectively. Each graphs contains two figures:

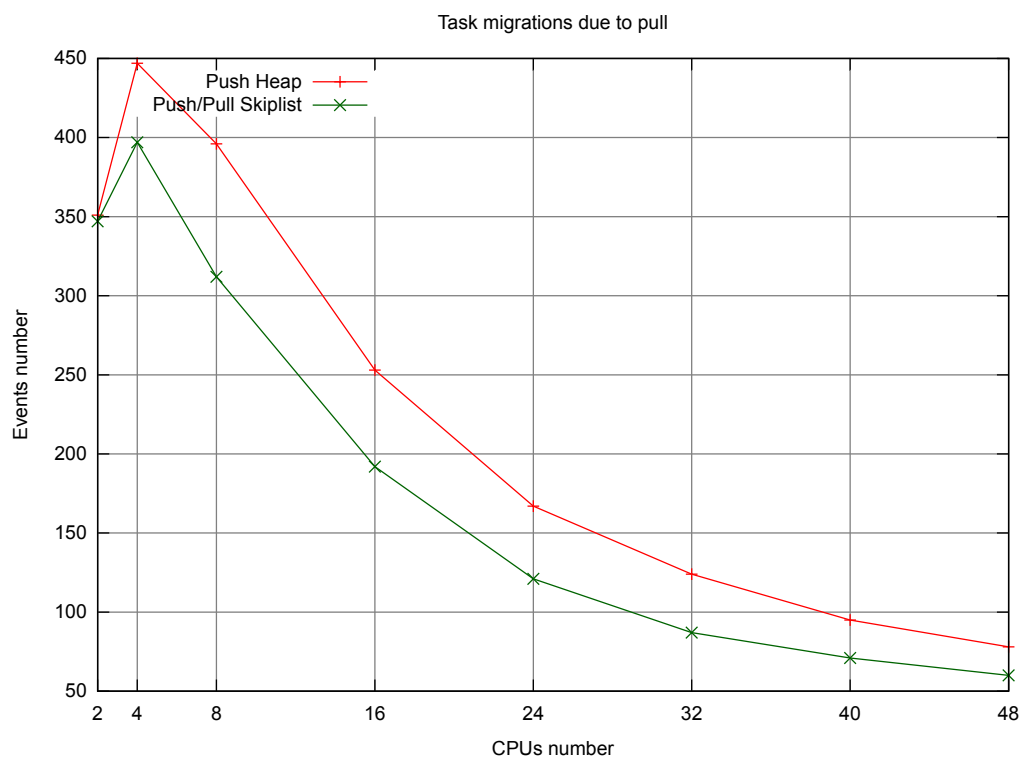


Figure 5.12: Number of task migrations due to pull operation

the *find* operation results on the top half and the *set* operation results on the bottom half.

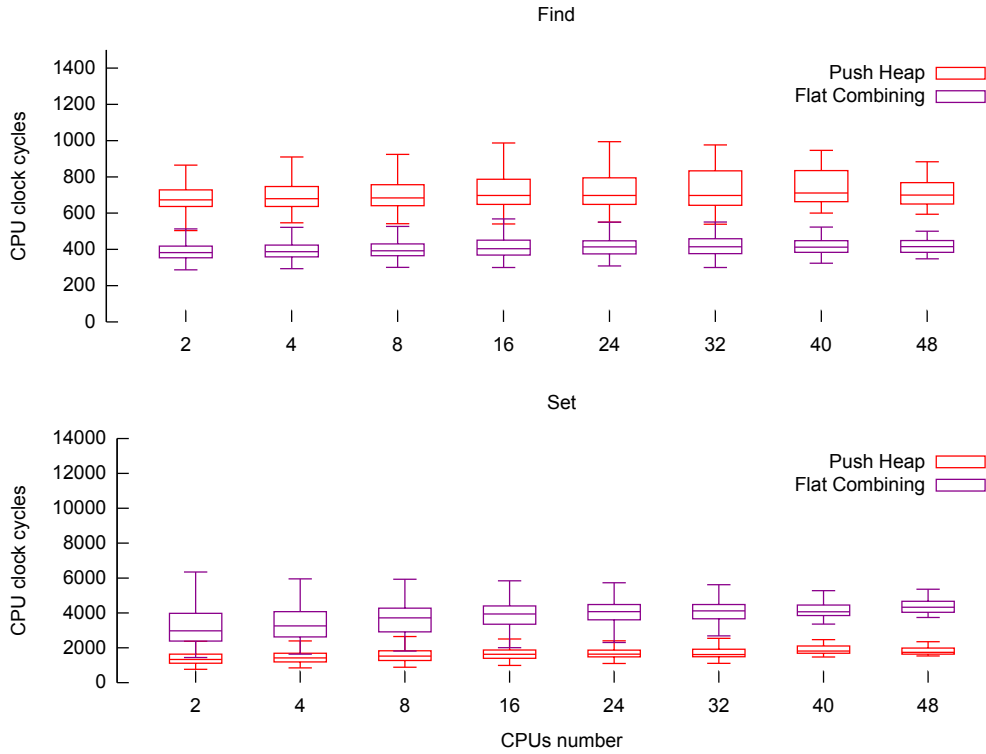


Figure 5.13: Bitmap flat combining push performance

Regarding the push operation, we compared the bitmap flat combining with the best current solution: the max-heap. We can see that, as with the skip list, flat combining reaches very high performance in the *find* operation. This is due to the best CPU cached value: if the cache is valid, we can immediately return that CPU index, so the operation is very fast. The *set* operation is instead slower for the bitmap flat combining solution. More importantly, we can see that the performance doesn't scale as well as with the max-heap: with an increasing number of CPUs, the spread between the two solutions is even more evident. With 48 online CPUs, the bitmap flat combining overcomes the 4000 CPU cycles threshold, while the max-heap remains under 2000 CPU cycles.

Regarding the pull operation, the trend is the same for both *find* and

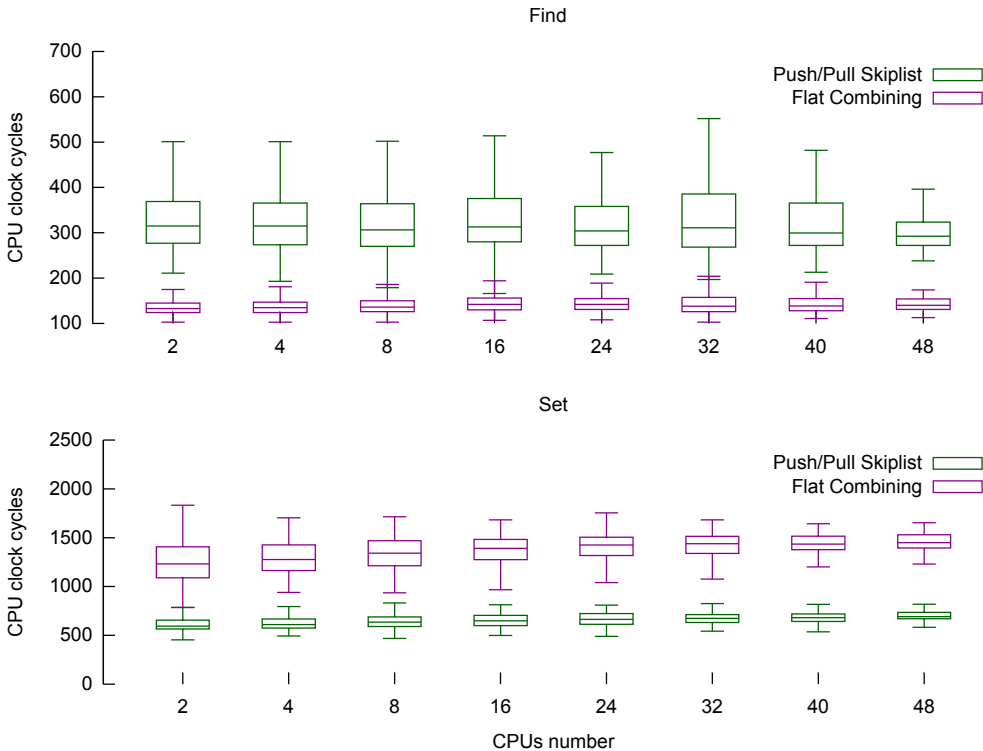


Figure 5.14: Bitmap flat combining pull performance

set operations. Here we have to point out that the comparison is done with the skip list as the improved pull algorithm has been tested with such data structure.

In conclusion, we see how the cache mechanism, initially introduced to keep the *cpull* updated among the underlying runqueues status, makes the solution very fast for the *find* operation, however, the flat combining framework is not adequate for the *set* operation. If we consider the results for the single-lock skip list another time, we can see how flat combining is even worse than that. This means that the underlying mechanism to defer work on the data structure puts a non negligible overhead.

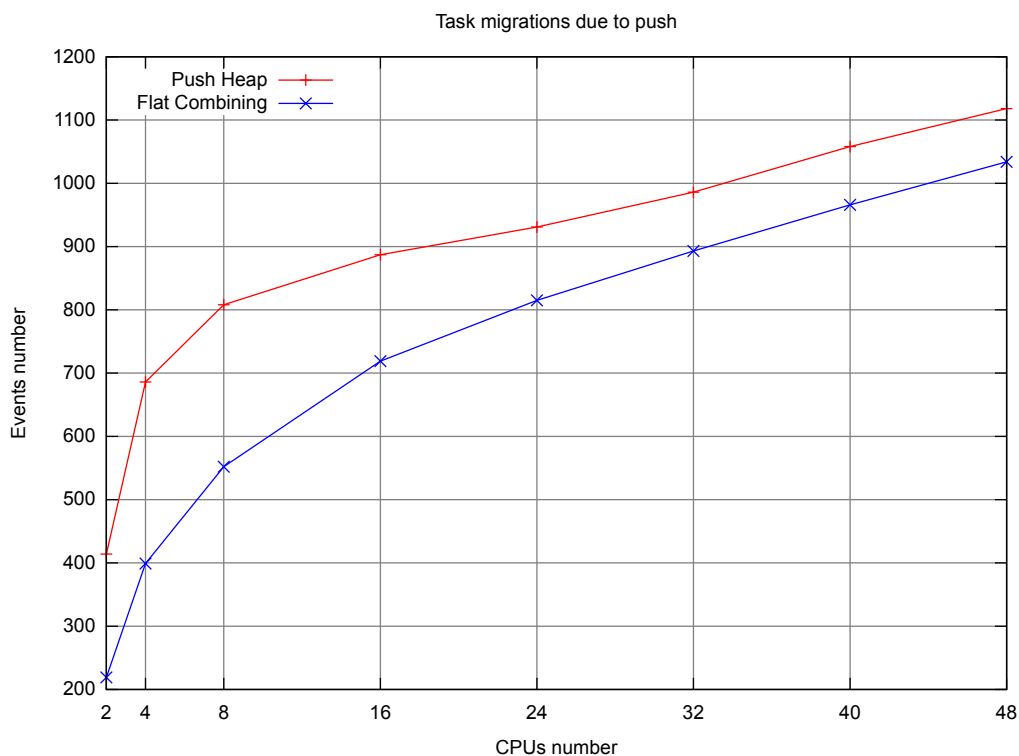


Figure 5.15: Number of successful push operations

Another insightful observation can be made referring to Figure 5.15, where the number of successful per CPU push operations is showed. In the graph the max-heap and the flat combining are compared. As we can see, the latter has a lower number of successful migrations: since we did

not change the push mechanism, then the work deferring mechanism is the responsible. Hence, the data structure cannot correctly represents the run-queues status under certain conditions. This is a notable drawback that highlights the inadequacy of this implementation.

5.6 Fastcache performance

As discussed in the previous sections, a good solution that aims at speeding up the task migration mechanism, has to achieve very high performance in the *set* operation. We have seen how the cache introduced with flat combining offers good performance in the *find* operation, but the way this cache is filled after being invalidated overcomes that benefit. With fastcache (see Section 3.4), we try to focus on the cache mechanism, refilling that with a very light-weight algorithm while ensuring that the *set* operation related work will be done immediately.

In Figure 5.16 we can see the performance of the *find* (top half) and *set* (bottom half) push related operations.

As usual, we compare fastcache with the max-heap, the fastest solution for the push operation. As we can see, fastcache overcomes the previous algorithm, both in the *find* and the *set* operations. The graph related to the latter operation is the most important: we can see that the trend remains flat as the number of the CPUs increases. In fact, the spread between the two graphs is more and more evident increasing the number of CPUs. Considering the 48 CPUs results, we can see how fastcache performs the *set* operation in about 600 CPU cycles, while the max-heap takes more then the 1500 CPU cycles.

The results for the pull operation, showed in Figure 5.17, are quite similar: we see how fastcache, compared to the skip list, tends to be always faster.

Also here we can see that the trend of the *set* operation is not as flat as the analogous one for push. This phenomenon can be explained looking at the number of successful migration due to push and pull, respectively, as shown in Figures 5.18 and 5.19.

We can see that the number of task migrations due to the push operation,

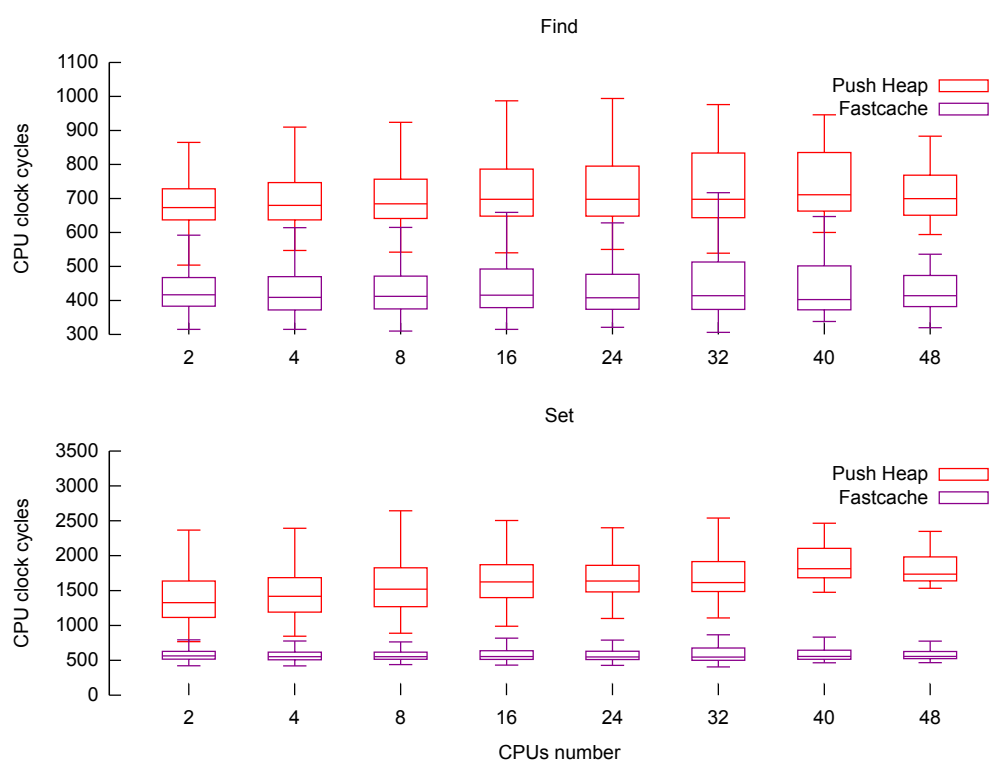


Figure 5.16: Fastcache push performance

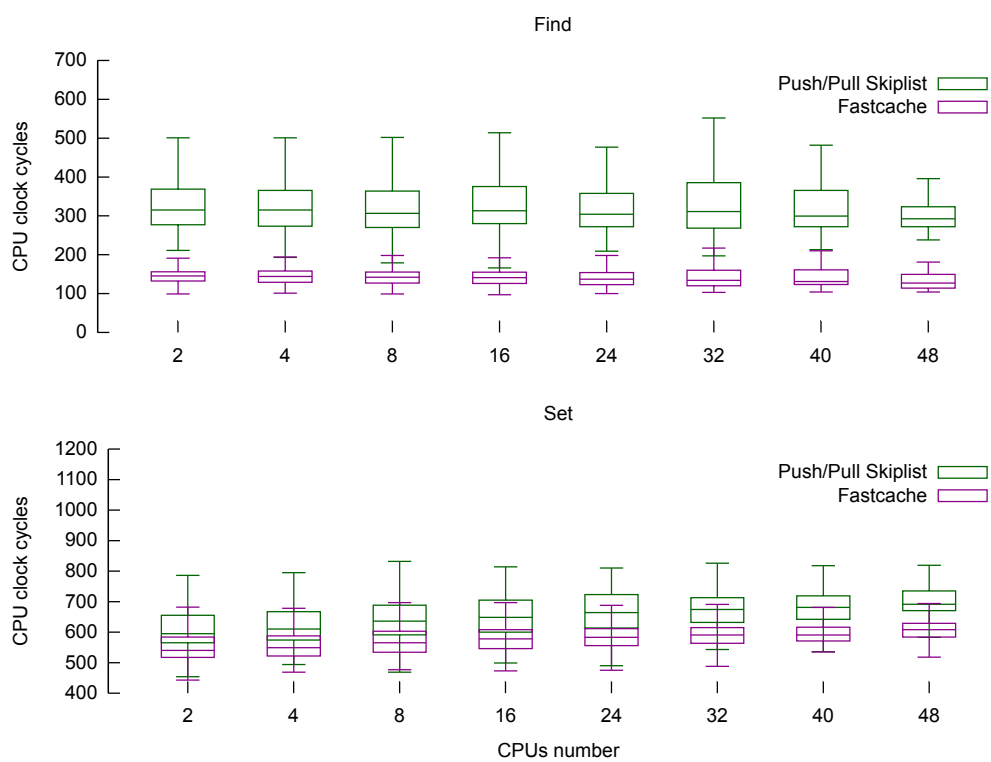


Figure 5.17: Fastcache pull performance

presented in the former figure, greatly overcomes the pull related one, in the latter figure.

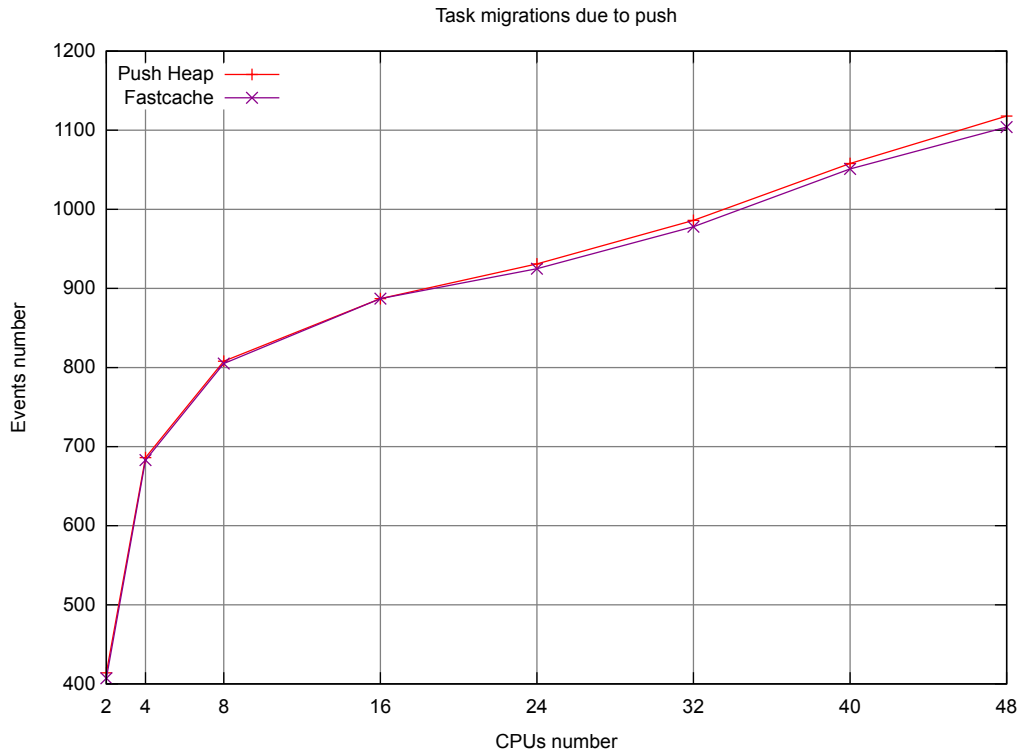


Figure 5.18: Number of task migrations due to push operation

Recall from Figures 1.5 and 3.2 how the push and pull operations work: the former has to cope with the `curr` deadline tasks, while the latter keeps track of the next deadline tasks. This means that is easier, for the pull `cpudl` data structure, to have an higher number of empty items: that is, CPUs with no next deadline tasks. This condition leads to an higher percentage of `set` operations with the flag `is_valid` set to zero, thus an higher percentage of cache invalidations. Since the slow path has to be followed more frequently for the pull related `set` operation, fastcache tends to be slightly dependent on the number of underlying CPUs. However, we can see that with an higher number of CPUs fastcache performs better than the skip list. So we can state that fastcache scales better than the skip list.

Finally, from these latest figures, we can see that the number of task

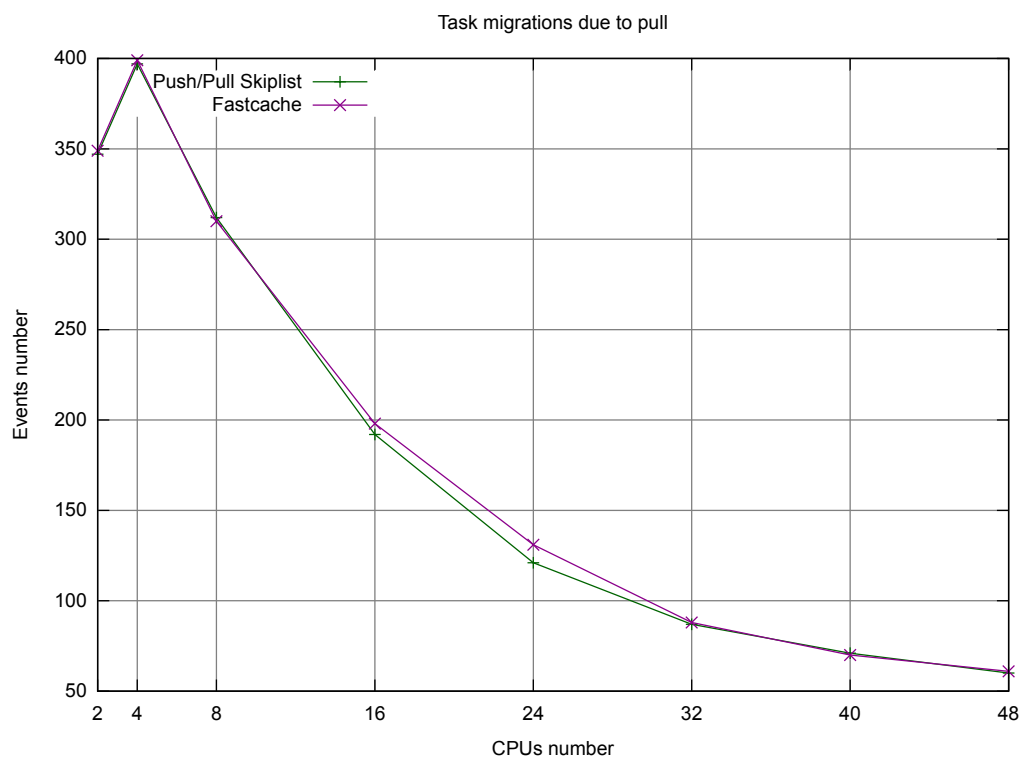


Figure 5.19: Number of task migrations due to pull operation

migrations is about the same for max-heap, skip list and fastcache.

Chapter 6

Conclusions and Future Work

In this thesis, we presented PRACTISE a tool for performance analysis and testing of real-time multicore schedulers for the Linux kernel. PRACTISE enables fast prototyping of real-time multicore scheduling mechanisms, allowing easy debugging and testing of such mechanisms in user-space. We showed the ability of this novel framework to predict the relative performance of multiple solutions.

Thanks to PRACTISE we were able to develop a set of innovative solutions to manage the task migrations in `SCHED_DEADLINE` scheduling class. We started with a probabilistic data structure, the skip list, that performs very well in *find* operation. Then we developed a specific implementation of the flat combining framework, named bitmap flat combining. This algorithm performs even better than the skip list in *find* operation. However, we showed that bitmap flat combining is not suitable for task migration mechanism. Finally, we developed fastcache, a novel algorithm that overcomes all previous one.

Regarding PRACTISE, a lot of improvements can be made.

First, it is possible to refine the framework adherence to the Linux kernel. In doing so, we have to enhance task affinity management, local run-queues capabilities and provide the possibility to generate random scheduling events following probability distributions gathered from real task sets execution traces. Moreover, an improvement to the *performance analysis* mode

can be made. In particular, the main goal is to alleviate the unpredictable latency introduced by a preemptive kernel while the user space code is under measure. A possible solution is first to develop some scripts that can translate the code in the kernel space equivalent one. After that, it will be possible to run that code inside PRACTISE, if the tool will be designed to work as a kernel module. Doing so, it will be possible to obtain more control on kernel preemption so as to obtain more accurate measurements.

Thanks to PRACTISE, we developed a set of improved solutions for the *cpudl* data structure. Driven by PRACTISE results, we decided to port each one of them in kernel space. The results show that, thanks to the *fastcache* algorithm, the task migration latency has been reduced, with a significant improvement from the point of view of the scalability.

A considerable result has also been obtained with the new pull algorithm: it has been showed that using a *cpudl* data structure even in the pull operation reduces the spurious task migrations, leading to a schedule closer to the theoretic *G-EDF*.

Regarding those aspects the research is all but over. To better understand how the schedules imposed by `SCHED_DEADLINE` are close to *G-EDF* a deep analysis is needed: it would be appropriate to develop a tool aimed to verify, for a given task sets, that the schedule obtained is really the *G-EDF* one. Driven by this results, it will be possible to address all cases where a schedule divergency arises.

Finally, the *fastcache* algorithm opens several usage scenarios that deserve to be investigated. Probably, the most interesting of those is the implementation of a single global (but scalable) ready queue: this way, reaching a real *G-EDF* scheduling policy will be plain. The original `SCHED_RT` scheduling class authors stated that a distributed runqueues design can scales well compared to a single global runqueue one [24]. But with the introduction of proper lock-free data structures and algorithms this statement may no longer be true.

Appendix A

Code listings

A.1 cpudl skip list implementation

```
/*
 * kernel/sched/cpudl.h
 *
 * CPU deadlines global management
 *
 * Author: Fabio Falzoi <fabio.falzoi@alice.it>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.
 */

#ifndef LINUX_CPUDLH
#define LINUX_CPUDLH

#include <linux/sched.h>

#define CPUDL_MAX_LEVEL 8
#define IDX_INVALID -1

struct skiplist_item {
    u64 dl;
    int level;
    struct skiplist_item *next[CPUDL_MAX_LEVEL];
    struct skiplist_item *prev[CPUDL_MAX_LEVEL];
    int cpu;
};
```



```

struct cpudl {
    raw_spinlock_t lock;
    struct skiplist_item *head;
    struct skiplist_item *cpu_to_idx[NR_CPUS];
    unsigned int level;
    cpumask_var_t free_cpus;
    bool (*cmp_dl)(u64 a, u64 b);
};

#ifdef CONFIG_SMP
int cpudl_find(struct cpudl *cp, struct cpumask *dlo_mask,
              struct task_struct *p, struct cpumask *later_mask);
void cpudl_set(struct cpudl *cp, int cpu, u64 dl, int is_valid);
int cpudl_init(struct cpudl *cp, bool (*cmp_dl)(u64 a, u64 b));
void cpudl_cleanup(struct cpudl *cp);
#else
#define cpudl_set(cp, cpu, dl) do { } while (0)
#define cpudl_init() do { } while (0)
#endif /* CONFIG_SMP */

#endif /* LINUX_CPUDL_H */

```

```

/*
 * kernel/sched_cpudl.c
 *
 * Global CPU deadlines management
 *
 * Author: Fabio Falzoi <fabio.falzoi@alice.it>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.
 */

#include <linux/gfp.h>
#include <linux/slab.h>
#include <linux/time.h>

#include "cpudl.h"

#define LEVEL_PROB_VALUE 0.20
#define NOT_IN_LIST -1
#define CPUDLRAND_MAX 10
#define CPUDL_HEAD_IDX -1

static inline u64 cpudl_detach(struct cpudl *list, struct skiplist_item *p)
{
    int i;

```

```

    for (i = 0; i <= p->level; i++) {
        p->prev[i]->next[i] = p->next[i];
        if (p->next[i])
            p->next[i]->prev[i] = p->prev[i];
    }

    while (!list->head->next[list->level] && list->level > 0)
        list->level--;

    p->level = NOT_IN_LIST;

    return p->dl;
}

static u64 cpudl_remove_idx(struct cpudl *list, const int cpu)
{
    struct skiplist_item *p;

    p = list->cpu_to_idx[cpu];

    if (p->level == NOT_IN_LIST)
        return 0;

    cpumask_set_cpu(cpu, list->free_cpus);

    return cpudl_detach(list, p);
}

static inline unsigned int cpudl_rand_level(unsigned int max)
{
    unsigned int level = 0, sorted;
    struct timespec limit;

    max = max > CPUDL_MAX_LEVEL - 1 ? CPUDL_MAX_LEVEL - 1 : max;

    do {
        level++;
        limit = current_kernel_time();
        sorted = ((unsigned int)limit.tv_nsec % CPUDL_RAND_MAX);
    } while ((sorted >= (((float)(1 - LEVEL_PROB_VALUE)) * CPUDL_RAND_MAX)) &&
            level < max);

    return level;
}

static int cpudl_insert(struct cpudl *list, const int cpu, u64 dl)
{
    struct skiplist_item *p;
    struct skiplist_item *update[CPUDL_MAX_LEVEL];

```

```

struct skiplist_item *new_node;
int cmp_res, level, i;
unsigned int rand_level;

new_node = list->cpu_to_idx[cpu];

new_node->dl = dl;

p = list->head;
level = list->level;
while(level >= 0) {
    update[level] = p;

    if(!p->next[level]) {
        level--;
        continue;
    }

    cmp_res = list->cmp_dl(new_node->dl, p->next[level]->dl);

    if(cmp_res > 0)
        p = p->next[level];
    else
        level--;
}

rand_level = cpudl_rand_level(list->level + 1);
new_node->level = rand_level;

if(rand_level > list->level)
    update[++list->level] = list->head;

for(i = 0; i <= rand_level; i++) {
    new_node->next[i] = update[i]->next[i];
    update[i]->next[i] = new_node;
    new_node->prev[i] = update[i];
    if(new_node->next[i])
        new_node->next[i]->prev[i] = new_node;
}

cpumask_clear_cpu(cpu, list->free_cpus);

return 0;
}

/*
 * cpudl_find - find the best (later-dl) CPU in the system
 * @cp: the cpudl skiplist context
 * @dlo_mask: mask of overloaded runqueues in the root domain (not used)
 * @p: the task

```

```

* @later_mask: a mask to fill in with the selected CPUs (or NULL)
*
* Returns: int - best CPU (skiplist maximum if suitable)
*/
int cpudl_find(struct cpudl *cp, struct cpumask *dlo_mask,
              struct task_struct *p, struct cpumask *later_mask)
{
    struct skiplist_item *first;
    u64 first_dl;
    int first_cpu, best_cpu = -1;
    const struct sched_dl_entity *dl_se;

    if(p)
        dl_se = &p->dl;

    if(later_mask && cpumask_and(later_mask, cp->free_cpus,
                                &p->cpus_allowed) && cpumask_and(later_mask,
                                later_mask, cpu_active_mask)) {
        best_cpu = cpumask_any(later_mask);
    } else {
        first = cp->head->next[0];
        if(!first)
            return -1;

        first_cpu = first->cpu;
        first_dl = first->dl;

        /*
         * if cpudl_find is called on behalf of
         * a pull attempt, we can not do any other
         * check, so we return immediately
         * the CPU value from cpudl structure
         */
        if(!p)
            return first_cpu;

        if(cpumask_test_cpu(first_cpu, &p->cpus_allowed) &&
           cp->cmp_dl(dl_se->deadline, first_dl)) {
            best_cpu = first_cpu;
            if(later_mask)
                cpumask_set_cpu(best_cpu, later_mask);
        }
    }

    return best_cpu;
}

/*
 * cpudl_set - update the cpudl skiplist
 * @cp: the cpudl skiplist context

```

```

* @cpu: the target cpu
* @dl: the new earliest deadline for this cpu
*
* Notes: assumes cpu_rq(cpu)->lock is locked
*
* Returns: (void)
*/
void cpudl_set(struct cpudl *cp, int cpu, u64 dl, int is_valid)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&cp->lock, flags);

    cpudl_remove_idx(cp, cpu);
    if(is_valid)
        cpudl_insert(cp, cpu, dl);

    raw_spin_unlock_irqrestore(&cp->lock, flags);
}

/*
* cpudl_init - initialize the cpudl structure
* @cp: the cpudl skiplist context
* @cmp_dl: function used to order deadlines inside structure
*/
int cpudl_init(struct cpudl *cp, bool (*cmp_dl)(u64 a, u64 b))
{
    int i;

    memset(cp, 0, sizeof(*cp));
    cp->cmp_dl = cmp_dl;
    raw_spin_lock_init(&cp->lock);
    cp->head = (struct skiplist_item *)kmalloc(sizeof(*cp->head), GFP_KERNEL)
        ;
    memset(cp->head, 0, sizeof(*cp->head));
    cp->head->cpu = CPUDL_HEAD_IDX;

    memset(cp->cpu_to_idx, 0, sizeof(*cp->cpu_to_idx) * NR_CPUS);

    for(i = 0; i < NR_CPUS; i++) {
        cp->cpu_to_idx[i] = (struct skiplist_item *)kmalloc(sizeof(*cp->
            cpu_to_idx[i]), GFP_KERNEL);
        memset(cp->cpu_to_idx[i], 0, sizeof(*cp->cpu_to_idx[i]));
        cp->cpu_to_idx[i]->level = NOT_IN_LIST;
        cp->cpu_to_idx[i]->cpu = i;
    }

    if(!alloc_cpumask_var(&cp->free_cpus, GFP_KERNEL))
        return -ENOMEM;
    cpumask_setall(cp->free_cpus);
}

```

```

    return 0;
}

/*
 * cpudl_cleanup - clean up the cpudl structure
 * @cp: the cpudl skiplist context
 */
void cpudl_cleanup(struct cpudl *cp)
{
    int i;

    for (i = 0; i < NR_CPUS; i++)
        kfree(cp->cpu_to_idx[i]);

    kfree(cp->head);
}

```

A.2 cpudl bitmap flat combining implementation

```

/*
 * kernel/sched/cpudl.h
 *
 * CPU deadlines global management
 *
 * Author: Fabio Falzoi <fabio.falzoi@alice.it>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.
 */

#ifndef LINUX_CPUDLH
#define LINUX_CPUDLH

#include <linux/sched.h>
#include <linux/threads.h>
#include <linux/cpumask.h>
#include <linux/types.h>

#include "bm_fc.h"

#define CPUDL_MAXLEVEL 8
#define NOT_IN_LIST -1
#define CPUDL_RAND_MAX ~0UL

```

```

#define CPUDL_HEAD_IDX    -1
#define NO_CACHED_CPU    -1

struct skiplist_item {
    u64 dl;
    int level;
    struct skiplist_item *next[CPUDL_MAX_LEVEL];
    struct skiplist_item *prev[CPUDL_MAX_LEVEL];
    int cpu;
};

struct cpudl {
    struct skiplist_item *head;
    struct skiplist_item *cpu_to_item[NR_CPUS];
    unsigned int level;

    cpumask_var_t free_cpus;

    bool (*cmp_dl)(u64 a, u64 b);

    struct flat_combining *fc;
};

#ifdef CONFIG_SMP
int cpudl_find(struct cpudl *cp, struct cpumask *dlo_mask,
              struct task_struct *p, struct cpumask *later_mask);
void cpudl_set(struct cpudl *cp, int cpu, u64 dl, int is_valid);
int cpudl_init(struct cpudl *cp, bool (*cmp_dl)(u64 a, u64 b));
void cpudl_cleanup(struct cpudl *cp);
#else
#define cpudl_set(cp, cpu, dl) do { } while (0)
#define cpudl_init() do { } while (0)
#endif /* CONFIG_SMP */

#endif /* _LINUX_CPUDL_H */

```

```

/*
 * kernel/sched_cpudl.c
 *
 * Global CPU deadlines management
 *
 * Author: Fabio Falzoi <falzoi@tecip.sssup.it>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.
 */

```

```

#include <linux/gfp.h>
#include <linux/slab.h>
#include <linux/time.h>
#include <linux/atomic.h>
#include <linux/atomic.h>
#include <asm/barrier.h>

#include "bm_fc.h"
#include "cpudl.h"

static inline u64 cpudl_detach(struct cpudl *list, struct skiplist_item *p)
{
    int i;

    for(i = 0; i <= p->level; i++) {
        p->prev[i]->next[i] = p->next[i];
        if(p->next[i])
            p->next[i]->prev[i] = p->prev[i];
    }

    while(!list->head->next[list->level] && list->level > 0)
        list->level--;

    p->level = NOT_IN_LIST;

    return p->dl;
}

static u64 cpudl_remove_idx(struct cpudl *list, const int cpu)
{
    struct skiplist_item *p;

    p = list->cpu_to_item[cpu];

    if(p->level == NOT_IN_LIST)
        return 0;

    cpumask_set_cpu(cpu, list->free_cpus);

    return cpudl_detach(list, p);
}

static inline unsigned int cpudl_rand_level(unsigned int max)
{
    unsigned int level = 0, sorted;
    struct timespec limit;

    max = max > CPUDL_MAXLEVEL - 1 ? CPUDL_MAXLEVEL - 1 : max;

    do {

```



```

    level++;
    limit = current_kernel_time();
    sorted = ((unsigned int)limit.tv_nsec % CPUDLRAND_MAX);
} while((sorted >= ((float)(1 - LEVELPROB_VALUE)) * CPUDLRAND_MAX)) &&
    level < max);

return level;
}

static int cpudl_insert(struct cpudl *list, const int cpu, u64 dl)
{
    struct skiplist_item *p;
    struct skiplist_item *update[CPUDL_MAX_LEVEL];
    struct skiplist_item *new_node;
    int cmp_res, level, i;
    unsigned int rand_level;

    new_node = list->cpu_to_item[cpu];

    new_node->dl = dl;

    p = list->head;
    level = list->level;
    while(level >= 0) {
        update[level] = p;

        if(!p->next[level]) {
            level--;
            continue;
        }

        cmp_res = list->cmp_dl(new_node->dl, p->next[level]->dl);

        if(cmp_res > 0)
            p = p->next[level];
        else
            level--;
    }

    rand_level = cpudl_rand_level(list->level + 1);
    new_node->level = rand_level;

    if(rand_level > list->level)
        update[++list->level] = list->head;

    for(i = 0; i <= rand_level; i++) {
        new_node->next[i] = update[i]->next[i];
        update[i]->next[i] = new_node;
        new_node->prev[i] = update[i];
        if(new_node->next[i])

```

```

        new_node->next[i]->prev[i] = new_node;
    }

    cpumask_clear_cpu(cpu, list->free_cpus);

    return 0;
}

static void cpudl_dispatcher(void *list, int cpu, u64 dline, int is_valid)
{
    struct cpudl *cp = (struct cpudl *)list;

    cpudl_remove_idx(cp, cpu);

    if(is_valid)
        cpudl_insert(cp, cpu, dline);
}

/*
 * cpudl_find - find the best (later-dl) CPU in the system
 * @cp: the cpudl skiplist context
 * @dlo_mask: mask of overloaded runqueues in the root domain (not used)
 * @p: the task
 * @later_mask: a mask to fill in with the selected CPUs (or NULL)
 *
 * Returns: int - best CPU (skiplist maximum if suitable)
 */
int cpudl_find(struct cpudl *cp, struct cpumask *dlo_mask,
              struct task_struct *p, struct cpumask *later_mask)
{
    u64 first_dl;
    int first_cpu, best_cpu = -1;
    const struct sched_dl_entity *dl_se;

    /*
     * for push operation, first we
     * search a suitable cpu in
     * cp->free_cpus (free CPUs mask),
     * otherwise we ask a cpu index
     * from cpudl
     */
    if(p)
        dl_se = &p->dl;
    if(later_mask && cpumask_and(later_mask, cp->free_cpus,
                               &p->cpus_allowed) && cpumask_and(later_mask,
                               later_mask, cpu_active_mask)) {
        best_cpu = cpumask_any(later_mask);
    } else {
        /*
         * we read best cpu from

```

```

    * flat combining cache
    */
    first_cpu = atomic_read(&cp->fc->cached_cpu);
    if(first_cpu < 0)
        return -1;
    else
        first_dl = (u64)atomic64_read(&cp->fc->current_dl[first_cpu]);
    /*
    * if cpudl_find is called on behalf of
    * a pull attempt, or first_cpu is equal
    * to -1, we can not do anything,
    * so we return immediately
    * the CPU value from cpudl structure
    */
    if(!p)
        return first_cpu;

    /*
    * if cpudl_find is called for
    * a push we must check the cpus_allowed
    * mask and the deadline
    */
    if(cpumask_test_cpu(first_cpu, &p->cpus_allowed) &&
        cp->cmp_dl(dl_se->deadline, first_dl)) {
        best_cpu = first_cpu;
        if(later_mask)
            cpumask_set_cpu(best_cpu, later_mask);
    }
}

return best_cpu;
}

/*
* cpudl_set - update the cpudl skiplist
* @cp: the cpudl skiplist context
* @cpu: the target cpu
* @dl: the new earliest deadline for this cpu
*
* Notes: assumes cpu_rq(cpu)->lock is locked
*
* Returns: (void)
*/
void cpudl_set(struct cpudl *cp, int cpu, u64 dl, int is_valid)
{
    struct pub_record *rec;
    int now_cached_cpu;
    u64 now_cached_dl = 0;

    /*

```

```

    * if is_valid is set we may have
    * to update the cached CPU
    */
if(is_valid) {
    /* we update immediately our deadline */
    atomic64_set(&cp->fc->current_dl[cpu], dl);
    while(1) {
        now_cached_cpu = atomic_read(&cp->fc->cached_cpu);
        if(now_cached_cpu != NO_CACHED_CPU)
            now_cached_dl = (u64)atomic64_read(&cp->fc->current_dl[
                now_cached_cpu]);
        /*
        * check if we have to update cached CPU value
        * we break the loop if the value must not be
        * updated or if we have to and the update succeed
        */
        if((now_cached_cpu != NO_CACHED_CPU &&
            now_cached_cpu != cpu &&
            cp->cmp_dl(now_cached_dl, dl)) ||
            atomic_cmpxchg(&cp->fc->cached_cpu, now_cached_cpu, cpu) == cpu)
            break;
    }
}

/*
 * if is_valid is clear we may have
 * to clear the cached CPU
 */
if(!is_valid) {
    /* we update immediately our deadline */
    atomic64_set(&cp->fc->current_dl[cpu], 0);
    while(1) {
        now_cached_cpu = atomic_read(&cp->fc->cached_cpu);
        if(now_cached_cpu != NO_CACHED_CPU)
            now_cached_dl = (u64)atomic64_read(&cp->fc->current_dl[
                now_cached_cpu]);
        if((now_cached_cpu != NO_CACHED_CPU && now_cached_cpu != cpu) ||
            atomic_cmpxchg(&cp->fc->cached_cpu, now_cached_cpu, cpu) == cpu)
            break;
    }
}

rec = fc_get_record(cp->fc, cpu);
rec->req = SET;
rec->par.set_p.cpu = cpu;
rec->par.set_p.dline = dl;
rec->par.set_p.is_valid = is_valid;
rec->h.set_h.function = cpudl_dispatcher;
fc_publish_record(cp->fc, cpu);

```

```

    fc_try_combiner(cp->fc);
}

/*
 * cpudl_init - initialize the cpudl structure
 * @cp: the cpudl skiplist context
 * @cmp_dl: function used to order deadlines inside structure
 */
int cpudl_init(struct cpudl *cp, bool (*cmp_dl)(u64 a, u64 b))
{
    int i;

    memset(cp, 0, sizeof(*cp));
    cp->cmp_dl = cmp_dl;

    cp->fc = fc_create(cp);
    atomic_set(&cp->fc->cached_cpu, NO_CACHED_CPU);

    cp->head = (struct skiplist_item *)kmalloc(sizeof(*cp->head), GFP_KERNEL)
        ;
    memset(cp->head, 0, sizeof(*cp->head));
    cp->head->cpu = CPUDL_HEAD_IDX;

    memset(cp->cpu_to_item, 0, sizeof(*cp->cpu_to_item) * NR_CPUS);
    for(i = 0; i < NR_CPUS; i++) {
        cp->cpu_to_item[i] = (struct skiplist_item *)kmalloc(sizeof(*cp->
            cpu_to_item[i]), GFP_KERNEL);
        memset(cp->cpu_to_item[i], 0, sizeof(*cp->cpu_to_item[i]));
        cp->cpu_to_item[i]->level = NOT_IN_LIST;
        cp->cpu_to_item[i]->cpu = i;
    }

    if(!alloc_cpumask_var(&cp->free_cpus, GFP_KERNEL))
        return -ENOMEM;
    cpumask_setall(cp->free_cpus);

    return 0;
}

/*
 * cpudl_cleanup - clean up the cpudl structure
 * @cp: the cpudl skiplist context
 */
void cpudl_cleanup(struct cpudl *cp)
{
    int i;

    for(i = 0; i < NR_CPUS; i++)
        kfree(cp->cpu_to_item[i]);
    kfree(cp->head);
}

```

```
    fc_destroy(cp->fc);
}



---



/*
 * kernel/sched/bm_fc.h
 *
 * Bitmap Flat Combining header file
 *
 * Author: Fabio Falzoi <fabio.falzoi@alice.it>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.
 */

#ifndef __BM_FC_H
#define __BM_FC_H

#include <linux/bitops.h>

/* flat combining parameters */

/*
 * no more than 32 publication
 * records allowed in this
 * implementation
 */
#define PUB.RECORD.PER.CPU    10

/* data structure operations type */
typedef enum {
    SET
} op_type;

/* data structure operations parameters */
typedef struct {
    int cpu;
    u64 dline;
    int is_valid;
} set_params;

typedef union {
    set_params set_p;
} params;

/* data structure operations handler */
typedef struct {
```

```

    void (*function)(void *data_structure, int cpu, u64 dline, int is_valid);
} set_handler;

typedef union {
    set_handler set_h;
} handler;

/* publication record */
struct pub_record {
    /* operation type */
    op_type req;
    /* operation parameters */
    params par;
    /* operation handler */
    handler h;
};

/* data structure lock interface */
#define DSLOCKED 1
#define DSUNLOCKED 0

struct data_structure_lock {
    atomic_t lock;
};

/* publication record list */
struct pub_list {
    /* publisher CPUs bitmap */
    u64 cpu_bitmap;
    /* active publication records bitmap */
    u32 rec_bitmap[NR_CPUS];
    /* publication record array */
    struct pub_record rec_array[NR_CPUS * PUB_RECORD_PER_CPU];
    /* last used per CPU publication record index */
    int last_used_idx[NR_CPUS];
};

/* flat combining helper structure */
struct flat_combining {
    /* concurrent data structure */
    void *data_structure;
    /* publication list */
    struct pub_list map;
    /* data structure lock */
    struct data_structure_lock ds_lock;
    /* cache cpu */
    atomic_t cached_cpu;
    /* dl array */
    atomic64_t current_dl[NR_CPUS];
};

```

```

/* flat combining interface */
struct flat_combining *fc_create(void *data_structure);

int fc_destroy(struct flat_combining *fc);

struct pub_record *fc_get_record(struct flat_combining *fc, const int cpu);

void fc_publish_record(struct flat_combining *fc, const int cpu);

/*
 * if we use a totally asynchronous flat combining
 * implementation, this function is going to be
 * used only internally in this module.
 * Otherwise, when we want to stop deferring work,
 * we have to call it explicitly.
 */
void fc_try_combiner(struct flat_combining *fc);

/*
 * if we want to ensure that a certain operation
 * will be executed synchronously and sequentially
 * we have to acquire and further release lock
 * on data structure with these functions
 */
void fc_data_structure_lock(struct flat_combining *fc);

void fc_data_structure_unlock(struct flat_combining *fc);

/* helper function useful for debugging purpose */
void fc_print_publication_list(struct flat_combining *fc);

#endif /* __BM_FC_H */

```

```

/*
 * kernel/sched/bm_fc.h
 *
 * Bitmap Flat Combining source file
 *
 * Author: Fabio Falzoi <fabio.falzoi@alice.it>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.
 */

#include <linux/kernel.h>
#include <linux/smp.h>

```



```
#include <linux/gfp.h>
#include <linux/slab.h>
#include <linux/threads.h>
#include <linux/bitops.h>
#include <linux/atomic.h>
#include <asm/barrier.h>

#include "bm_fc.h"

/* bitmap management helper functions */
inline void bitmap64_set(u64 *bitmap, int n)
{
    *bitmap |= ((u64)1 << n);
    smp_wmb();
}

inline void bitmap32_set(u32 *bitmap, int n)
{
    *bitmap |= ((u32)1 << n);
    smp_wmb();
}

inline void bitmap64_clear(u64 *bitmap, int n)
{
    *bitmap &= ~((u64)1 << n);
    smp_wmb();
}

inline void bitmap32_clear(u32 *bitmap, int n)
{
    *bitmap &= ~((u32)1 << n);
    smp_wmb();
}

inline int bitmap64_test(u64 *bitmap, int n)
{
    smp_rmb();
    return ((*bitmap & ((u64)1 << n)) > (u64)0);
}

inline int bitmap32_test(u32 *bitmap, int n)
{
    smp_rmb();
    return ((*bitmap & ((u32)1 << n)) > (u32)0);
}

inline int bitmap64_fls(u64 *bitmap)
{
    smp_rmb();
    return fls64(*bitmap) - 1;
}
```

```

}

inline int bitmap32_fls(u32 *bitmap)
{
    smp_rmb();
    return fls(*bitmap) - 1;
}

/* data structure lock interface */
void fc_lock(struct data_structure_lock *ds_lock)
{
    int old, ret;

    while(1) {
        smp_rmb();
        old = atomic_read(&ds_lock->lock);
        if(old == DSLOCKED)
            continue;
        /*
         * Any atomic operation that modifies
         * some state in memory and returns information
         * about the state implies an SMP-conditional
         * general memory barrier on each side of the
         * actual operation
         *
         * See Documentation/memory-barriers.txt for
         * further details
         */
        ret = atomic_cmpxchg(&ds_lock->lock, old, DSLOCKED);
        if(ret == old)
            break;
    }
}

int fc_trylock(struct data_structure_lock *ds_lock)
{
    int old, ret;

    smp_rmb();
    old = atomic_read(&ds_lock->lock);
    if(old == DSLOCKED)
        return -1;
    ret = atomic_cmpxchg(&ds_lock->lock, old, DSLOCKED);
    if(ret == old)
        return 0;
    else
        return -1;
}

void fc_unlock(struct data_structure_lock *ds_lock)

```

```

{
    /*
     * Since atomic_set() doesn't returns
     * anything about new or old memory state
     * we have to issue a memory barrier
     */
    atomic_set(&ds_lock->lock, DS_UNLOCKED);
    smp_wmb();
}

/* flat combining interface */
static void fc_do_combiner(struct flat_combining *fc)
{
    struct pub_list *map = &fc->map;
    struct pub_record *rec;
    int cpu_index, rec_index;

    while((cpu_index = bitmap64_fls(&map->cpu_bitmap)) >= 0) {
        while((rec_index = bitmap32_fls(&map->rec_bitmap[cpu_index])) >= 0) {
            rec = &map->rec_array[cpu_index * PUB_RECORD_PER_CPU + rec_index];
            switch(rec->req) {
                case SET:
                    rec->h.set_h.function(fc->data_structure,
                                         rec->par.set_p.cpu,
                                         rec->par.set_p.dline,
                                         rec->par.set_p.is_valid);
                    break;
                default:
                    printk(KERN_ERR "ERROR: unknown operation type on cpu %d pub
                                record\n", cpu_index);
            }
            bitmap32_clear(&map->rec_bitmap[cpu_index], rec_index);
        }
        bitmap64_clear(&map->cpu_bitmap, cpu_index);
    }
}

struct flat_combining *fc_create(void *data_structure)
{
    struct flat_combining *fc;

    fc = (struct flat_combining *)kmalloc(sizeof(*fc), GFP_KERNEL);
    memset(fc, 0, sizeof(*fc));
    atomic_set(&fc->ds_lock.lock, DS_UNLOCKED);
    smp_wmb();
    fc->data_structure = data_structure;

    return fc;
}

```

```

int fc_destroy(struct flat_combining *fc)
{
    if(fc) {
        kfree(fc);
        return 0;
    }

    return -1;
}

struct pub_record *fc_get_record(struct flat_combining *fc, const int cpu)
{
    struct pub_list *map = &fc->map;
    int idx_to_use;

    /* next publication record to use */
    idx_to_use = map->last_used_idx[cpu];

    while(1) {
        /* if record is not busy we use it */
        if(!bitmap32_test(&map->rec_bitmap[cpu], idx_to_use))
            return &map->rec_array[cpu * PUB_RECORD_PER_CPU + idx_to_use];

        /*
         * no free record, so:
         * we set our bit in cpu_bitmap and
         * we spin to become a combiner
         */
        while(bitmap32_test(&map->rec_bitmap[cpu], idx_to_use)) {
            bitmap64_set(&map->cpu_bitmap, cpu);
            fc_try_combiner(fc);
        }
    }
}

void fc_publish_record(struct flat_combining *fc, const int cpu)
{
    struct pub_list *map = &fc->map;
    int idx_to_use;

    idx_to_use = map->last_used_idx[cpu];
    map->last_used_idx[cpu] = (map->last_used_idx[cpu] + 1) %
        PUB_RECORD_PER_CPU;

    bitmap32_set(&map->rec_bitmap[cpu], idx_to_use);
    bitmap64_set(&map->cpu_bitmap, cpu);
}

void fc_try_combiner(struct flat_combining *fc)
{

```

```

    if(!fc_trylock(&fc->ds_lock)) {
        fc_do_combiner(fc);
        fc_unlock(&fc->ds_lock);
    }
}

void fc_data_structure_lock(struct flat_combining *fc)
{
    fc_lock(&fc->ds_lock);
}

void fc_data_structure_unlock(struct flat_combining *fc)
{
    fc_unlock(&fc->ds_lock);
}

void fc_print_publication_list(struct flat_combining *fc)
{
    struct pub_list *map = &fc->map;
    int i, cpu = smp_processor_id();

    trace_printk("[%d] - CPUs map: %llu\n", cpu, map->cpu_bitmap);
    for(i = 0; i < NR_CPUS; i++)
        trace_printk("[%d] - cpu %d map %u\n", cpu, i, map->rec_bitmap[i]);
}

```

A.3 cpudl fastcache implementation

```

/*
 * kernel/sched/cpudl.h
 *
 * CPU deadlines global management
 *
 * Author: Fabio Falzoi <fabio.falzoi@alice.it>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.
 */

#ifndef LINUX_CPUDLH
#define LINUX_CPUDLH

#include <linux/cpumask.h>
#include <linux/types.h>

#define CACHE_LINE_SIZE 64

```

```

struct curr_dl_item {
    atomic64_t dl;
    u8 padding[CACHE_LINE_SIZE - sizeof(atomic64_t)];
};

struct cpudl {
    cpumask_var_t free_cpus;

    bool (*cmp_dl)(u64 a, u64 b);

    atomic_t cached_cpu;
    struct curr_dl_item current_dl[NR_CPUS] __attribute__((aligned (
        CACHE_LINE_SIZE)));

    raw_spinlock_t lock;
};

#ifdef CONFIG_SMP
int cpudl_find(struct cpudl *cp, struct cpumask *dlo_mask,
              struct task_struct *p, struct cpumask *later_mask);
void cpudl_set(struct cpudl *cp, int cpu, u64 dl, int is_valid);
int cpudl_init(struct cpudl *cp, bool (*cmp_dl)(u64 a, u64 b));
void cpudl_cleanup(struct cpudl *cp);
#else
#define cpudl_set(cp, cpu, dl) do { } while (0)
#define cpudl_init() do { } while (0)
#endif /* CONFIG_SMP */

#endif /* _LINUX_CPUDL_H */

```

```

/*
 * kernel/sched/cpudl.c
 *
 * CPU deadlines global management
 *
 * Author: Fabio Falzoi <fabio.falzoi@alice.it>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.
 */

#include <linux/sched.h>
#include <linux/types.h>
#include <asm/barrier.h>
#include <linux/spinlock.h>
#include <linux/gfp.h>

```

```

#include <linux/slab.h>
#include <linux/cpumask.h>

#include "cpudl.h"

/* cache not valid */
#define NO_CACHED_CPU    -1
/* no cpu with deadline task */
#define NO_CPU_DL        -2
/* not valid dl */
#define NO_CACHED_DL     0

static inline void update_cache_slow(struct cpudl *cp)
{
    int best_cpu = NO_CPU_DL;
    u64 best_dl = NO_CACHED_DL;
    u64 current_dl;
    int i;

    if(!cpumask_full(cp->free_cpus))
        for_each_cpu_not(i, cp->free_cpus) {
            current_dl = (u64)atomic64_read(&cp->current_dl[i].dl);
            if(current_dl == NO_CACHED_DL)
                continue;
            if(best_dl == NO_CACHED_DL ||
                cp->cmp_dl(best_dl, current_dl)) {
                best_dl = current_dl;
                best_cpu = i;
            }
        }

    smp_wmb();
    atomic_set(&cp->cached_cpu, best_cpu);
}

/*
 * cpudl_find - find the best CPU in the system
 * @cp: the cpudl context
 * @dlo_mask: mask of overloaded runqueues in the
 * root domain (used only for push operation)
 * @p: the task
 * @later_mask: a mask to fill in with the selected
 * CPUs (or NULL)
 *
 * Returns: int - best CPU to/from migrate
 * the task
 */
int cpudl_find(struct cpudl *cp, struct cpumask *dlo_mask,
               struct task_struct *p, struct cpumask *later_mask)
{

```

```

int now_cached_cpu = NO.CACHED.CPU;
u64 now_cached_dl;
unsigned long flags;
int best_cpu = -1;
const struct sched_dl_entity *dl_se;

if (later_mask && cpumask_and(later_mask, cp->free_cpus,
    &p->cpus_allowed) && cpumask_and(later_mask,
    later_mask, cpu_active_mask))
    return cpumask_any(later_mask);

now_cached_cpu = atomic_read(&cp->cached_cpu);
if(now_cached_cpu == NO.CPU.DL || now_cached_cpu == NO.CACHED.CPU)
    return -1;

/*
 * cpudl_find is called on behalf
 * of a pull, so we don't care about
 * cp->current_dl[now_cached_cpu] value
 */
if(!p)
    return now_cached_cpu;

/*
 * if cpudl_find is called on behalf of
 * a push we must check the cpus_allowed
 * mask and the deadline
 *
 * A read barrier is needed,
 * otherwise we may see
 * cp->cached_cpu updated
 * with an old value in
 * cp->current_dl
 */
smp_rmb();
now_cached_dl = (u64)atomic64_read(&cp->current_dl[now_cached_cpu].dl);
/*
 * a parallel operation may have
 * changed the deadline value of
 * now_cached_cpu
 */
if(now_cached_dl == NO.CACHED.DL)
    return -1;

dl_se = &p->dl;
if(cpumask_test_cpu(now_cached_cpu, &p->cpus_allowed) &&
    cp->cmp_dl(dl_se->deadline, now_cached_dl)) {
    best_cpu = now_cached_cpu;
    if(later_mask)
        cpumask_set_cpu(best_cpu, later_mask);
}

```



```

    }

    return best_cpu;
}

/*
 * cpudl_set - update the cpudl skiplist
 * @cp: the cpudl skiplist context
 * @cpu: the target cpu
 * @dl: the new earliest deadline for this cpu
 *
 * Notes: assumes cpu_rq(cpu)->lock is locked
 *
 * Returns: (void)
 */
void cpudl_set(struct cpudl *cp, int cpu, u64 dl, int is_valid)
{
    int now_cached_cpu;
    u64 now_cached_dl;
    bool updated = false;
    unsigned long flags;

    /*
     * if is_valid is set we may have
     * to update the cached CPU
     */
    if(is_valid) {
        cpumask_clear_cpu(cpu, cp->free_cpus);
        atomic64_set(&cp->current_dl[cpu].dl, dl);
        while(1) {
            now_cached_cpu = atomic_read(&cp->cached_cpu);
            if(now_cached_cpu != NO_CACHED_CPU &&
                (now_cached_cpu != cpu || updated)) {
                smp_rmb();
                now_cached_dl = (u64)atomic64_read(&cp->current_dl[now_cached_cpu].
                    dl);
            } else {
                if(!raw_spin_trylock_irqsave(&cp->lock, flags)) {
                    update_cache_slow(cp);
                    raw_spin_unlock_irqrestore(&cp->lock, flags);
                    updated = true;
                }
                continue;
            }
        }

        if((now_cached_cpu != NO_CPU_DL &&
            now_cached_dl != NO_CACHED_DL &&
            cp->cmp_dl(dl, now_cached_dl)) ||
            atomic_cmpxchg(&cp->cached_cpu, now_cached_cpu, cpu) ==
                now_cached_cpu)

```

```

        break;
    }
} else {
    cpumask_set_cpu(cpu, cp->free_cpus);
    atomic64_set(&cp->current_dl[cpu].dl, NO_CACHED_DL);
    /*
     * if is_valid is clear we may have
     * to clear the cached CPU
     */
    while(1) {
        now_cached_cpu = atomic_read(&cp->cached_cpu);
        if(now_cached_cpu == cpu &&
            atomic_cmpxchg(&cp->cached_cpu, now_cached_cpu, NO_CACHED_CPU) !=
                now_cached_cpu)
            continue;
        if(now_cached_cpu == NO_CACHED_CPU) {
            if(!raw_spin_trylock_irqsave(&cp->lock, flags)) {
                update_cache_slow(cp);
                raw_spin_unlock_irqrestore(&cp->lock, flags);
            }
            /*
             * here we doesn't have
             * to wait for the cache to
             * be valid, so we can
             * exit immediately
             */
        }
        break;
    }
}
}

/*
 * cpudl_init - initialize the cpudl structure
 * @cp: the cpudl skiplist context
 * @cmp_dl: function used to order deadlines inside structure
 */
int cpudl_init(struct cpudl *cp, bool (*cmp_dl)(u64 a, u64 b))
{
    int i;

    raw_spin_lock_init(&cp->lock);

    atomic_set(&cp->cached_cpu, NO_CACHED_CPU);
    for(i = 0; i < NR_CPUS; i++)
        atomic64_set(&cp->current_dl[i].dl, NO_CACHED_DL);

    cp->cmp_dl = cmp_dl;

    if(!alloc_cpumask_var(&cp->free_cpus, GFP_KERNEL))

```

```

    return -ENOMEM;
    cpumask_setall(cp->free_cpus);

    return 0;
}

/*
 * cpudl_cleanup - clean up the cpudl structure
 * @cp: the cpudl skiplist context
 */
void cpudl_cleanup(struct cpudl *cp)
{
    free_cpumask_var(cp->free_cpus);
}

```

A.4 Improved pull algorithm

```

/*
 * Deadline Scheduling Class (SCHED_DEADLINE)
 *
 * Earliest Deadline First (EDF) + Constant Bandwidth Server (CBS).
 *
 * Tasks that periodically executes their instances for less than their
 * runtime won't miss any of their deadlines.
 * Tasks that are not periodic or sporadic or that tries to execute more
 * than their reserved bandwidth will be slowed down (and may potentially
 * miss some of their deadlines), and won't affect any other task.
 *
 * Copyright (C) 2012 Dario Faggioli <raistlin@linux.it>,
 *                    Juri Lelli <juri.elli@gmail.com>,
 *                    Michael Trimarchi <michael@amarulasolutions.com>,
 *                    Fabio Checconi <fabio@gandalf.sssup.it>
 */

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)))
        goto out;

    cpu = cpudl_find(&this_rq->rd->pull_cpudl, this_rq->rd->dlo_mask, NULL,
                    NULL);
    if(cpu == -1 || this_cpu == cpu)
        goto out;
}

```

```

src_rq = cpu_rq(cpu);

/* Might drop this_rq->lock */
double_lock_balance(this_rq, src_rq);

/*
 * If the pullable task is no more on the
 * runqueue, we're done with it
 */
if(src_rq->dl.dl_nr_running <= 1)
    goto skip;

p = pick_next_earliest_dl_task(src_rq, this_cpu);

/*
 * We found a task to be pulled if:
 * - p can run on this cpu (otherwise pick_next_earliest_dl_task has
 *   returned NULL)
 * - it preempts our current (if there's one)
 */
if (p && (!this_rq->dl.dl_nr_running ||
    dl_time_before(p->dl.deadline, this_rq->dl.earliest_dl.curr))) {

    WARN_ON(p == src_rq->curr);
    WARN_ON(!p->on_rq);

    /*
     * Then we pull iff p has actually an earlier
     * deadline than the current task of its runqueue.
     */
    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);

}

skip:
    double_unlock_balance(this_rq, src_rq);
out:
    return ret;
}

```

Acknowledgments

Vorrei innanzi tutto ringraziare il prof. Giuseppe Lipari per avermi dato l'occasione di svolgere questa tesi. Il suo continuo supporto stato un fondamentale aiuto per raggiungere il risultato finale. Ma non solo: durante il periodo di tesi mi stata anche data l'occasione di presentare parte del lavoro al Workshop OSPERT 2012. E' stata un'esperienza molto impegnativa ma, al contempo, molto gratificante.

Desidero ringraziare anche il prof. Paolo Ancilotti per la sua disponibilit nell'assistermi durante la discussione della tesi.

Un ringraziamento particolare va a Juri Lelli. Juri mi ha assistito per tutta la durata del lavoro, aiutandomi in ogni aspetto tecnico della tesi: senza la sua grande disponibilit non potrei scrivere adesso queste parole.

Adesso, il momento di ringraziare famiglia ed amici. Non semplice condensare in poche parole tutto quello che vorrei esprimere. Il primo pensiero va sicuramente a mia nonna e a Katia.

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] G. Buttazzo. *Sistemi in Tempo Reale*. Pitagora Editrice Bologna, 2006.
- [6] 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.
- [7] Johnathan Corbet. Cfs group scheduling. <http://lwn.net/Articles/240474/>.
- [8] Johnathan Corbet. Scheduling domains. <http://lwn.net/Articles/80911/>.

- [9] 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.
- [10] Linux documentation. Design of the cfs scheduler. Documentation/scheduler/sched-design-CFS.txt.
- [11] L. Dozio and P. Mantegazza. Real-time distributed control using rtai. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '03)*, Hakodate, Hokkaido, Japan, May 2003.
- [12] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, Brussels, Belgium, July 2010.
- [13] Community Research European Commission. Sixth framework programme. http://ec.europa.eu/research/fp6/index_en.cfm.
- [14] D. Faggioli, M. Trimarchi, and F. Checconi. Sched_deadline. <https://github.com/jlelli/sched-deadline>.
- [15] FRESCOR. Framework for real-time embedded systems based on contracts. <http://www.frescor.org/index.php?page=FRESCOR-homepage>.
- [16] P. Gerum. The xenomai project, implementing a rtos emulation framework on gnu/linux. Nov. 2002.
- [17] PREEMPT_RT group. Config_preempt_rt patch set. <http://www.kernel.org/pub/linux/kernel/projects/rt/>.
- [18] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. *Work*, 2010.

- [19] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2011)*. July 2011.
- [20] G. Lipari and C. Scordino. Linux and real-time: Current approaches and future oppostunities. *IEEE International Congress ANIPLA*, 2006.
- [21] LITMUS^{RT}. Linux testbed for multiprocessor scheduling in real-time systems. <http://www.cs.unc.edu/~anderson/litmus-rt/>.
- [22] 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.
- [23] Paul E Mckenney. Memory Barriers: a Hardware View for Software Hackers. 2009.
- [24] Ingo Molnar. Modular scheduler core and completely fair scheduler [cfs]. <http://lkml.org/lkml/2007/4/13/180>.
- [25] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 Instruction Set Architectures. Intel White Paper, September 2010.
- [26] OCERA Project. Open components for embedded real-time applications. <http://www.ocera.org/index.html>.
- [27] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Elements*.
- [28] William Pugh. Concurrent Mainteneace of Skip Lists. 1989, 1990.
- [29] 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.

- [30] RTAI. Rtai - the realtime application interface for linux from diapi.
<https://www.rtai.org/>.
- [31] 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.
- [32] 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.
- [33] V. Yodaiken. The rtlinux manifesto. In *Proceeding of the Fifth Linux Expo*, Raleigh, North Carolina, Mar. 1999.