SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2335

Deadline scheduling in the Linux kernel

Juri Lelli[1], Claudio Scordino[2,*,†], Luca Abeni[3] and Dario Faggioli[1]

[1]*ReTiS Lab., Scuola S.Anna, p.zza Martiri della Libertà 33, 56127 Pisa, Italy*
[2]*Evidence Srl, Via Carducci 56, 56010 San Giuliano Terme, Pisa, Italy*
[3]*University of Trento, via Sommarive 9, 38123 Povo, Trento, Italy*

SUMMARY

During the last decade, there has been a considerable interest in using Linux in real-time systems, especially for industrial control. The simple and elegant design of Linux guarantees reliability and very good performance, while its open-source license allows to modify and change the source code according to the user needs. However, Linux has been designed to be a general-purpose operating system. Therefore, it presents some issues like unpredictable latencies and limited support for real-time scheduling. In this paper, we present our experience in the design and implementation of the real-time scheduler that has been recently included in the Linux kernel. The scheduler is based on the Resource Reservation paradigm, which allows to enforce temporal isolation between the running tasks. We describe the genesis of the project, the challenges we have encountered, the implementation details and the API offered to the programmers. Then, we show the experimental results measured on a real hardware. Copyright © 2015 John Wiley & Sons, Ltd.

Received 16 June 2014; Revised 8 May 2015; Accepted 12 May 2015

KEY WORDS: real-time; resource-reservation; scheduling; operating system; Linux

1. INTRODUCTION

Linux is a full-featured operating system (OS) kernel, originally designed to be used in server or desktop environments. Since then, Linux has evolved and grown to be used in almost all computer areas – among others, embedded systems and supercomputers. Thanks to its high modularity and scalability, in fact, Linux can run either on a small microcontroller and on a powerful machine with thousands of CPUs.

In the last years, there has been a considerable interest in using Linux for real-time control systems, from both academic institutions and industry. The main reason for this raising interest is that Linux is an open-source project. Its license (i.e. GNU General Public License) allows to access, modify and redistribute (even sell) the software as long as the recipient can exercise the same rights (access to the source code included). This way, a company is not tied to the OS provider anymore, and is free of accessing and modifying the OS at will. The open-source license also helped the growth of a large community of developers who continuously add new features and port Linux to new architectures. Nowadays, there is a huge amount of programs, libraries and tools available as open-source code that can be used to build a customized version of the OS. Another reason for this interest is the very large amount of hardware platforms and peripherals already supported by Linux. Therefore, using a Linux-based OS can likely reduce the total cost of a project (i.e. development, maintainance and licensing) with respect to a commercial real-time operating system (RTOS).

Unfortunately, the standard mainline Linux kernel is not adequate to be used in an RTOS, because it has been designed to be a general-purpose operating system kernel. Hence, the main design goal

*Correspondence to: Claudio Scordino, Evidence Srl, Via Carducci 56, 56010 San Giuliano Terme, Pisa, Italy.
[†]E-mail: claudio@evidence.eu.com

has concerned the optimization of the average throughput (i.e. the amount of 'useful work' performed by the system in the unit of time) instead of meeting the timing constraints of each running application. As a result, the standard Linux kernel does not provide good real-time performance (i.e. low kernel latencies) or other features needed for supporting real-time applications (for example, advanced scheduling / resource management algorithms).

In this paper, we describe a real-time scheduler for the CPU that we have recently included in Linux. The paper is organized as follows. In Section 2, we present the previous work about real-time on Linux. In Section 3, we describe the scheduling algorithm, the implementation details as well as the API offered to the programmer. Then, in Section 4, we provide some experimental results on a real hardware. Finally, in Section 5, we state our conclusions and we provide information about the future work.

2. RELATED WORK

In the last decade, several (sometimes orthogonal) mechanisms have been proposed to add real-time capabilities to the Linux kernel. These mechanisms can be grouped in the following classes [1].

Hardware Abstraction Layers (HALs). The first technique consists on reducing the kernel latency through a small abstraction layer (also called hypervisor), which virtualizes the hardware exposed to the Linux kernel [2]. This layer takes full control of the hardware resources (mainly, interrupts and system timers) and directly handles the real-time tasks at a higher priority than Linux. The Linux kernel is thus scheduled by the HAL as a low-priority task. The real-time applications cannot use functionalities provided by the Linux kernel, unless by temporarily loosing their real-time guarantees.

This approach, originally introduced by Finite State Machine Labs with real-time Linux (RTLinux) [3], then acquired by Wind River [4], is now provided by two community-driven projects. The Real-Time Application Interface (RTAI) project [5], started in 1997, has developed the Adeos kernel [6] as a replacement for the RTLinux core. RTAI has excellent latencies on the x86 architecture, but the support for other architectures is limited. Xenomai [7] – a fork of the RTAI project – relies on the Adeos core as well and supports a larger amount of embedded platforms, but it offers slightly less performance than RTAI [8].

Better pre-emption and interrupt handling strategies. The second technique consists on reducing the Linux kernel latencies by improving the strategies for pre-emption and interrupt handling. Ingo Molnar and Andrew Morton proposed the *Low Latency Patch* [9] to introduce explicit preemption points in specific blocks of code that may execute for long intervals of time. Robert Love proposed a different approach with Montavista's *Preemptible Kernel Patch* [10], making the kernel pre-emptible like the user-space. This patch, accepted into the mainline Linux kernel since version 2.6, allows a context switch even when another process is running in kernel context. Hence, it becomes possible to pre-empt a process at any point, as long as the kernel is in a consistent state (i.e. no lock is held). A comparison of the performance of the two patches has been provided by Clark Williams [11].

PREEMPT_RT [12, 13] is an open-source patch developed by the same developers of the mainline kernel (e.g. Steven Rostedt, Ingo Molnar, etc.). The patch introduces a set of orthogonal mechanisms to reduce kernel latency and make the kernel more deterministic. Some of these mechanisms (e.g. high resolution timers and priority inheritance) have been already merged into the mainline kernel. Among the mechanisms not yet merged, there are the re-implementation of spinlocks as rtmutexes (further reducing the parts of kernel code that cannot be pre-empted) and threaded interrupts (converting interrupt handlers into pre-emptible kernel threads with a tunable priority). PREEMPT_RT is widely used to create real-time versions of Linux. For example, RTDroid [14] is a recent project aiming at creating a real-time version of Android by patching the underlying Linux kernel with PREEMPT_RT. The real-time scheduler that we present in this paper can coexist with PREEMPT_RT and thus can help in further improving the real-time performance of projects like RTDroid.

DOI: 10.1002/spe

Scheduling and resource management algorithms. Several projects in the last decade have proposed real-time algorithms for the management of hardware resources in Linux. Well-known examples are Budget Fair Queueing (BFQ) [15] and BFQ+ [16] for disk operations (both merged into the mainline kernel) and Quick Fair Queueing [17] for the network. However, the major interest has traditionally been focused on the CPU, being the primary hardware resource.

Traditionally, Linux has offered simple POSIX-compliant real-time fixed-priority policies (i.e. SCHED_FIFO and SCHED_RR) that execute the highest-priority ready task until it explicitly blocks. Indeed, fixed-priority scheduling is an effective solution for executing tasks in embedded RTOSs, where the number of tasks is very small and the behaviour of each real-time task is perfectly known. Unfortunately, these policies are not optimal for real-time scheduling in General-Purpose Operating Systems (GPOSs) like Linux. The first reason is the missing isolation between the running tasks [18]. For example, if a regular non-privileged user is enabled to access the scheduling facilities, then he can starve the rest of the system by rising his task to the highest priority. Starvation, however, can also occur when the actual execution time of a high-priority task is longer than expected. The second reason is related to the interface provided by fixed-priority, which is not oriented to the application requirements. The timing requirements of the application, in fact, cannot be directly expressed, and must be translated into scheduling priorities, taking into account all tasks running in the system. In a GPOS, this means that whenever a new real-time task is created, the priorities of all other tasks must be checked and potentially reordered.

Additionally, the Linux kernel provides the control groups ('cgroups') mechanism to limit the amount of CPU time consumed by a SCHED_FIFO or SCHED_RR task. This mechanism is implemented using an algorithm similar to the deferrable server [19], which can generate scheduling anomalies when the tasks block and wake-up dynamically. For example, if a task is not ready for execution at the beginning of its scheduling period, then another task with a lower priority may experience a deadline miss when the higher-priority task will become ready again (the interested reader can refer to the simple example shown in Figure 2 of [18] – the example in that figure is based on an earliest deadline first (EDF) scheduler, but similar effects can happen using fixed priorities too). Because this kind of anomalies can happen even if the taskset is schedulable according to response time analysis [20] (or similar kinds of schedulability analysis), providing real-time guarantees to tasks scheduled by fixed-priorities with control groups in a dynamic scenario can be very difficult.

The scheduling policies of a real-time GPOS, instead, should provide full temporal isolation between the running tasks. This means that the worst-case temporal behaviour of a task (for example, measured by its worst-case response time) should not be affected by the behaviour of the other tasks running on the system. This way, if a task misbehaves trying to use a resource for a time longer than expected, it is stopped (or its priority is decreased) and cannot affect the real-time performance of the other tasks running in the system.

Moreover, the scheduling algorithm should expose a natural interface for expressing the application requirements in terms of timing constraints.

Resource Reservation is an effective way for providing temporal isolation in GPOSs. The first example of this technique has been the Capacity Reserve algorithm proposed by Mercer and Tokuda [21]. Then, several algorithms have been proposed in the literature. Notable examples are Constant Bandwidth Server (CBS) [22], Greedy Reclamation of Unused Bandwidth (GRUB) [23], HGRUB [24] and Idle-time Reclaiming Improved Server (IRIS) [25].

The Linux/RK project [26] was one of the first attempts of introducing Resource Reservation techniques in the Linux kernel, by adding both CPU (with fixed-priority scheduling) and disk (with EDF scheduling [27]) reservations. Unfortunately, the project has not been maintained throughout the course of the years, also because of the drastic changes made to the Linux kernel at the 2.4 and 2.6 releases.

To overcome the limitations of the existing Linux scheduler, Faggioli *et al.* proposed an implementation of the POSIX SCHED_SPORADIC real-time policy [28]. The scheduler was developed with the aim of being integrated into the mainstream kernel, by proposing a very limited set of modifications to the standard scheduler. The policy provided temporal isolation, and the implementation was integrated with the existing user-space cgroups interface [29]. The advantage of the scheduler was the semantics already standardized by POSIX. However, it still suffered the typical

DOI: 10.1002/spe

limitations of priority-based policies (i.e. API not oriented to timing requirements) [30]. Unfortunately, the patch did not receive enough interest from the kernel community, so its development was discontinued several years ago.

A real-time scheduler based on Resource Reservation for Linux was originally developed at Real-Time Systems Laboratory (ReTiS Lab) within the Open Components for Embedded Real-time Applications (OCERA) European project [18, 31]. Because the Linux kernel 2.4.18 did not yet provide a modular scheduling framework, the scheduler was developed as a loadable kernel module. A small patch for the Linux kernel exported the necessary symbols and the relevant events to the real-time scheduler. In practice, the standard Linux scheduler acted as a dispatcher for the external real-time scheduler. The original work implemented the CBS algorithm [22]. Next works implemented more advanced algorithms [32]. The original scheduler was then ported to kernel 2.6 through the Adaptive Quality of Service Architecture project [33], adding a user-level library for feedback scheduling. These projects always remained at a prototype level. The major drawbacks were the lack of multi-core support and the difficulty of porting to different kernel versions. However, these attempts allowed to investigate the advantages of a resource reservation scheduler in Linux.

The Linux Testbed for Multiprocessor Scheduling in Real-Time System (LITMUS^{RT}) [34] is a project originally born at the University of North Carolina, now developed at Max Planck Institute for Software Systems, to provide an experimental platform for real-time systems. The research is particularly focused on multiprocessor scheduling and synchronization. Like OCERA, the project relies on a real-time extension of the Linux kernel, which simplifies the prototyping of scheduling algorithms. The project has allowed investigating the behaviour and the performance of various scheduling algorithms in the Linux kernel (e.g. partitioned versus global scheduling policies [35]). However, currently, there are no plans to turn it into a production-quality system.

The ExSched project [36] went a step further by proposing a scheduling framework that does not need any modification to the Linux kernel. The core component is still a kernel module, which only depends on a few kernel primitives and communicates with user-level through a proper entry in the `dev` filesystem. The framework aims at being cross-platform and is also available for the VxWorks OS. Some experimental results showed that ExSched is less efficient than our scheduler, with more than 180% of overhead for low system loads [36].

Specific resource reservation scheduling policies for Linux have been developed also by Libenzi (with SCHED_SOFTRR [37]) and by Kolivas (with SCHED_ISO [38]). These policies allow non-privileged users to run tasks at real-time priority up to a maximum processor usage. Thus, it becomes possible to provide deterministic latencies to time-sensitive applications, still ensuring system stability and fairness.

More recently, Checconi *et al.* proposed real-time throttling [39]. The Interactive Realtime Multimedia Applications on Service Orientated Infrastructures scheduler is based on a direct modification of the POSIX real-time scheduling class and, in particular, to the throttling mechanism already existing. This mechanism has been enhanced to provide groups of real-time tasks not just a limit but also precise scheduling guarantees (in terms of a guaranteed runtime every period, on each of the available CPUs). The modification of the POSIX scheduler has been designed from scratch with Symmetric Multi-Processing (SMP) support in mind, and it implements a hierarchical scheduling policy based on both deadlines and priorities – that is, fixed-priority scheduling is nested inside EDF scheduling.

Although very valuable, all these projects have not been able to gain enough consensus and traction for being merged into the mainline kernel. However, part of the experience obtained during their development has been used in the development of the real-time scheduler SCHED_DEADLINE, which has been recently merged in the Linux kernel. It is important to highlight that, unlike some of the projects that we have mentioned, SCHED_DEADLINE is not a scheduling framework, but it is the implementation of a specific scheduling algorithm that will be described in the next sections. Of course, such implementation can be modified to implement different algorithms – for example, including CPU reclaiming [23, 40]. However, the comparison of the scheduling algorithm (i.e. CBS) with other algorithms is out of the scope of the paper, which aims at describing the algorithm accepted in the Linux kernel.

DOI: 10.1002/spe

3. SCHED_DEADLINE

Since the release of 2.6.23, Linux has a modular scheduling framework that can be easily extended with additional policies. It has been implemented by Molnar as a replacement of the previous O(1) scheduler. It consists of a main `schedule()` function and a set of scheduling classes, each encapsulating a specific scheduling policy. The binding between each policy and the related scheduler is done through a set of hooks (i.e. function pointers) provided by each scheduling class and called by the core scheduler.

SCHED_DEADLINE originally was born within the context of the ACTORS project, financially supported by the European Commission under the FP7 framework. ACTORS combined virtualization, feedback control and a data-flow programming model for realizing a high-performance infrastructure for adaptable resource management in embedded devices. The reference platform was a Linux-based multi-core ARM chip. Within this project, the partner Evidence Srl was in charge of the development of a real-time CPU scheduler at kernel level. The scheduler has been created in collaboration with the ReTiS Lab (which already had a wide experience in development of real-time schedulers for Linux).

Rather than making ad hoc changes to the Linux scheduling core (which has been proven to offer a lower overhead [41]), Evidence Srl created an additional in-kernel scheduling class to ease the acceptance of the scheduler into the official Linux kernel. The project, in fact, aimed at the ambitious goal of creating a real-time scheduler available by default on all Linux distributions. Thus, SCHED_DEADLINE sits on top of the Linux modular scheduling framework, without changing the behaviour of already existing scheduling policies (i.e. best-effort and fixed-priority). Figure 1 shows the set of scheduling policies now available on Linux. The counterpart is that SCHED_DEADLINE automatically inherited the performance and the latencies of the Linux scheduling framework.

The first version of the scheduler was presented to the Linux community in 2009 with the name of SCHED_EDF [42, 43]. The name was then changed to SCHED_DEADLINE after the request of the kernel community [44]. During the course of the years, the scheduler has been maintained and periodically submitted through the Linux kernel mailing list. As the popularity of the scheduler increased, a higher number of kernel developers provided their own contribution. Each release aligned the code to the latest version of the kernel and took into account the comments received at the previous submission. From a release to another, in fact, some features were added or removed, according to the requests of the kernel community. In particular – besides the change of the scheduler name – the kernel community asked the following changes:

- Implementation of the deadline inheritance protocol [45] for both mutexes and futexes;
- Implementation of task migration through the common push & pull kernel mechanism;
- Removal of the whole cgroups interface [29];
- Removal of bandwidth reclaiming (which, based on a particular flag, scheduled the task at a lower priority once the budget was exhausted).

Another sensitive topic subject of discussion has been the behaviour in case of `fork`. The initial design let this system call succeed by creating another process with SCHED_DEADLINE policy and

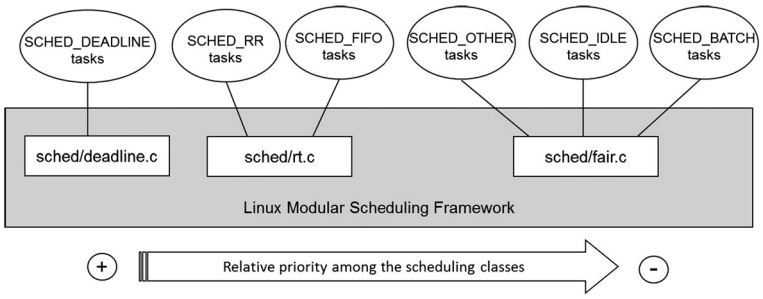

Figure 1. Linux modular scheduling framework.

DOI: 10.1002/spe

budget equal to 0. The kernel maintainers, instead, asked to be more conservative by making the `fork` system call fail for any SCHED_DEADLINE process.

Finally, after more than 4 years and the submission of nine releases, the patch has been accepted and merged into the Linux kernel 3.14. We will now describe the internal design and the API of the version that has been accepted.

3.1. Scheduling algorithm

SCHED_DEADLINE provides a scheduling policy suitable for real-time applications by implementing a resource reservation algorithm[‡] in the Linux kernel. This algorithm, named CBS, implements resource reservations over an EDF scheduler [27]. Resource Reservations [21] are an effective technique for providing temporal isolation, which allows to use real-time scheduling even in GPOSs. The basic idea is to assign each real-time task a 'reservation' (Q_i, T_i), meaning that the task is guaranteed to execute for a 'maximum runtime' Q_i[§] every period of time T_i. The goal of the scheduler is to schedule the tasks in such a way that each task receives the guaranteed CPU time (the runtime) Q_i in every period T_i. Then, if Q_i and T_i are properly sized, the timing constraints of the task are respected. If a task tries to execute for more than Q_i in a period T_i, then the scheduler throttles it (avoiding to select it from execution) until the end of the current reservation period. In this way, each task is constrained to not use more than its reserved share – that is, a maximum of Q_i every T_i units of time.

As mentioned, CBS [22] is a well-known algorithm implementing CPU Reservations. It belongs to the class of aperiodic servers with dynamic priorities, deriving inspiration from the service mechanisms proposed in the Dynamic Sporadic Server [46, 47] and the Total Bandwidth Server [48]. In particular, it uses dynamic priorities to correctly cope with dynamic aperiodic arrivals.

For each task, the scheduler keeps track of two additional state variables: the remaining runtime q_i (which represents the amount of remaining processor time that the task can use in the current reservation period of size T_i[¶]) and a scheduling deadline d_i, which is used to assign a dynamic priority to the task. When the CBS starts handling a task, q_i and d_i are initialized to 0.

The CBS algorithm works as follows. When a task wakes up at time t, the scheduler checks if the current scheduling deadline d_i can be used (if $q_i < (d_i - t)\frac{Q_i}{T_i}$), otherwise a new scheduling deadline $d_i = t + T_i$ is generated and the remaining runtime q_i is recharged to Q_i. While a task executes, its remaining runtime is decreased as $dq_i = -dt$ (this is often known as the *accounting rule*). When the remaining runtime reaches 0 ($q_i = 0$), the task is throttled[‖] and cannot be selected by the scheduler for execution (this is often known as the *enforcement rule*, because it forces the task to not consume more than Q_i time units in a period T_i). The task exits the throttled state only at time $t = d_i$, when the runtime is recharged to its maximum value ($q_i = Q_i$) and the scheduling deadline is postponed ($d_i = d_i + T_i$), decreasing the task's priority.

Notice that the CBS algorithm adds an additional task state (the throttled state, indicating that the task already consumed all of its runtime for the current period) to the three task states used by traditional schedulers (the *ready state*, indicating that a task is in the ready queue and can be selected for execution by the scheduler; the *running state*, indicating that a task has been selected by the scheduler; and the *blocked state*, indicating that a task is not in the ready queue – and hence cannot be selected for execution – because it is waiting for some event to happen). The resulting state transition diagram is shown in Figure 2. The interested reader can refer to the original paper [22] for an in-depth description of the algorithm.

Once the scheduling deadlines have been assigned, CBS orders the tasks according to the EDF [27] algorithm (i.e. from the earliest to the latest deadline). If the total utilization $\sum_i \frac{Q_i}{T_i}$ of the real-time tasks is smaller than a certain threshold, EDF can provide some service guarantees to the tasks. For example, on single-core systems (or on multi-core systems using partitioned scheduling) EDF

[‡]More specifically, a CPU reservation algorithm.
[§]Notice that the Linux 'runtime' is often called 'budget' in the real-time literature.
[¶]Notice that in the original CBS papers, this 'period' T_i is named 'server period' to avoid confusions with the task period.
[‖]In real-time literature, the word *depleted* is often used instead of throttled.

DOI: 10.1002/spe

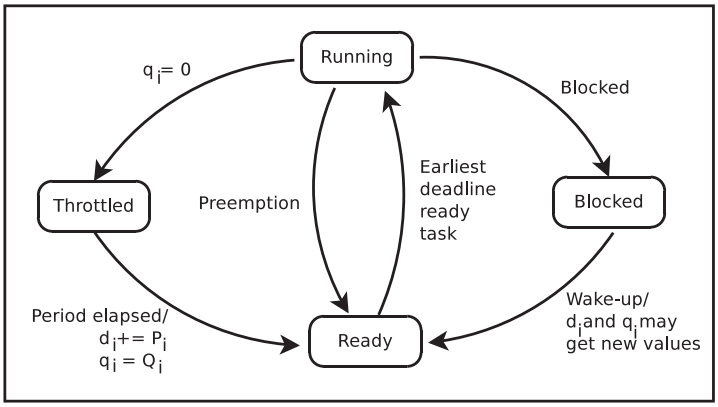

Figure 2. State transition diagram of the CBS algorithm.

guarantees that a set of n real-time tasks is schedulable (that is to say, every task is guaranteed to receive Q_i time units every T_i) if and only if

$$\left(\sum_{i=1}^{n} \frac{Q_i}{T_i}\right) \leqslant 1. \tag{1}$$

This check can be easily obtained through an admission control.

The previously mentioned temporal isolation property derives from these service guarantees. Notice that the admission test does not consider any interaction between tasks; hence, temporal isolation is only achieved between independent tasks.

When implemented in the Linux kernel, SCHED_DEADLINE improved the original CBS by allowing to specify a relative deadline D_i different from the period T_i: when a task wakes up at time t and a new scheduling deadline is generated (because $q_i \geqslant (d_i - t)\frac{Q_i}{T_i}$), it is set to $d_i = t + D_i$, while when the runtime is recharged the scheduling deadline is postponed by T_i : $d_i = d_i + T_i$. Thus, as illustrated in Section 3.4, the user-level API of SCHED_DEADLINE allows also to set this additional parameter.

3.2. Multiprocessor scheduling

When dealing with multiprocessor architectures, two different schemes exist for real-time scheduling: global and partitioned scheduling, namely. With global scheduling, all ready tasks are put in the same data structure and then run on the different cores. With partitioned scheduling, instead, a different list of ready tasks is maintained for each processor. Both approaches have pros and cons, and a proper discussion of which is the best scheme is out of the scope of this paper (the interested reader can refer to a vast literature, e.g. among others, Lelli *et al.* [49] and Bastoni *et al.* [35]).

The way Linux deals with multiprocessor scheduling is often called *distributed runqueue*: each processor has its own ready queue – as in partitioning – but tasks can migrate between the different queues. This way, it is possible to move tasks among the processors without the very high contention typical of global scheduling. Real-time scheduling policies perform migrations by means of so-called *push* and *pull* operations. A push operation is started by a source CPU when a task becomes available for execution on its runqueue (e.g. the task wakes up or resumes from the throttled state, or when a task is pre-empted by a higher-priority one). The operation searches all the runqueues to select the best one, where the task currently running has the lowest priority among all running tasks, and a lower priority than the task just activated. A pull operation, instead, is started by a target CPU when it is about to schedule a task of priority lower than the previously scheduled one (e.g. when a task is throttled or blocks). This operation scans every runqueue of the system to find a ready but not running backlogged task with higher priority than all local tasks. It has to be noted that both push and pull operations operate only on tasks that are not currently scheduled.

DOI: 10.1002/spe

In our scheduler, we have followed the same approach. We have implemented one runqueue for each processor, using a red-black tree data structure for efficient manipulation. We have also implemented the logics for migrating tasks among the runqueues. Therefore, SCHED_DEADLINE supports both partitioned scheduling (using the cpuset facility, see succeeding sections) and global scheduling (where the scheduler is left free to move tasks across CPUs). In the latter case, the logics assigns the m available processors to the m earliest deadline tasks. To speed-up push operations, we implemented a max-heap that keeps track of the deadlines of the currently scheduled tasks on each CPU, paired by a bitmask used to identify CPUs free of tasks belonging to the new scheduling policy [50].

The Linux kernel has a facility (named *cpuset*), which provides a mechanism for creating logical entities, called 'cpusets', that encompass definitions of CPUs and Memory Nodes (for Non-Uniform Memory Access (NUMA) systems). Cpusets constrain the CPU and memory placement of a task to only the resources defined within that cpuset. Sets of tasks can be assigned to these cpusets to constrain the resources that they use (e.g. the CPUs they can run on). The tasks can be moved from one cpuset to another to utilize other resources defined in those other cpusets. If the system is configured with cpusets containing single CPUs, when the user assigns a task to a specific cpuset, the scheduler behaves as a truly partitioned scheme with a very small overhead. The other mean by which a task can be restricted to execute on a subset of CPUs is called *task affinity*. Although Gujarati *et al.* already studied admission control for Arbitrary Processor Affinities [51], affinity setting is currently disabled for tasks handled by our scheduler because of the complexity of the admission control.

3.3. Shared resources

Tasks can access shared resources using mutually exclusive semaphores, often referred to as *mutex*es. The portion of code in a task between a lock and an unlock operation on a mutex is called *critical section*. If a task needs to enter a critical section, it may be blocked by the fact that another task has already locked the corresponding mutex: this latter task is called *lock owner* or *lock holder*. In real-time systems, it is important to compute for how long, in the worst case, a task may remain blocked on a lock.

A priority inversion happens when a task is blocked by a lower priority task. Without a proper protocol to control access to critical sections, the duration of the priority inversion may become too long, or even unbounded; for this reason, the priority inheritance protocol (PIP) [45] has been proposed as a simple and effective way to reduce priority inversion. According to the PIP, when a task is blocked on a lock, the lock owner inherits the highest between its priority and the priorities of the blocked tasks. In this way, it cannot be pre-empted by medium priority tasks, thus reducing the blocking time.

In our scheduler, we adopted a solution based on the very same idea. Because dynamic deadlines constitute tasks priorities, when a task blocks on a lock, the lock owner inherits the earliest deadline between its current deadline and the deadline of the blocked task (this approach is similar to Jansen *et al.* [52], that is in turn based on the work by Sha *et al.* [45]). In addition, if the deadline is inherited from the blocked task, the lock owner enters a boosted state. While in this state, the enforcement mechanism is modified: when the runtime is depleted, it is immediately refilled and the deadline is postponed by one period (notice that the priority of the task decreases, but the task is not throttled). This behaviour guarantees that tasks waiting for a lock owner to exit a critical section do not experience additional delays caused by the CBS throttled state (i.e. the interval of time during which the lock owner is throttled, waiting for a runtime refill). The obvious simplicity of this solution is however paid back by the possibility that some 'inconsistencies' may happen. In fact, because the lock owner inherits only the deadline of the blocked task, but not its budget, it actually consumes a budget under an unrelated deadline. Hence, the admission test should be modified to account for the blocking times caused by this deadline inheritance.

SCHED_DEADLINE currently handles shared resources through deadline inheritance because this is the simplest possible solution: because merging a complex scheduler in the mainline kernel is a long process and it is not possible to merge all the features at the same time, it has been decided

DOI: 10.1002/spe

to start by merging the simplest part, introducing new and more advanced features in a second time. Hence, the more appropriate approach for resource sharing will be merged later. Such a better approach consists in inheriting both budget and deadline of the blocked task, as in BWI [53]. Hence, as stated in Section 5, the final goal is merging an existing implementation of the (Multiprocessor) Bandwidth Inheritance (M-BWI) protocol [54].

3.4. User-level API

As explained, the SCHED_DEADLINE scheduling policy implements the CBS algorithm. The temporal isolation property provided by CBS can be used for two main reasons:

- Providing timing guarantees to hard or soft real-time tasks: each SCHED_DEADLINE task is guaranteed to receive the specified share of CPU time, regardless of the behaviour of the other tasks running on the system; these guarantees are important for time-sensitive applications (e.g. real-time control) that need to execute a certain amount of work within a timing constraint.
- Enforcing timing confinement: the task is forced to use no more than its assigned CPU share; this is useful to limit the amount of CPU used by high-priority tasks which, in case of bug or intensive computation, may starve the rest of the system. It is also useful in virtualization environments, for a fair allocation of the CPU among several virtual machines.

In the former case, the important property is temporal isolation from the other tasks, for the benefit of the task itself. In the latter case, instead, it is an isolation for the benefit of the other tasks. Again, notice that temporal isolation holds only for independent tasks; hence, when there are significant interactions between tasks (frequent invocation of 'heavy' system calls, shared resources, message passing, etc...) temporal confinement and timing guarantees might be compromised. When the BWI algorithm will be merged in SCHED_DEADLINE, these interferences will be limited within groups of interacting tasks.

We now describe the Application Programming Interface (API) provided to application developers. The existing system calls `sched_setscheduler()` and `sched_getscheduler()` have not been extended, because of the binary compatibility issues that modifying the `sched_param` data structure would have raised for existing applications. Therefore, two new system calls called `sched_setattr()` and `sched_getattr()` have been introduced. These syscalls also support the other existing scheduling policies – that is, the interpretation of the arguments depends on the selected policy. Therefore, they are expected to replace the previous system calls (which will be left to not break existing applications).

The prototype of these new system calls is shown in the following table:

```
#include <sched.h>

struct sched_attr {
    u32 size;
    u32 sched_policy;
    u64 sched_flags;

    /* SCHED_OTHER, SCHED_BATCH */
    s32 sched_nice;

    /* SCHED_FIFO, SCHED_RR */
    u32 sched_priority;

    /* SCHED_DEADLINE */
    u64 sched_runtime;
    u64 sched_deadline;
    u64 sched_period;
};

int sched_setattr(pid_t pid,
    const struct sched_attr *attr,
```


DOI: 10.1002/spe

```
    unsigned int flags);

int sched_getattr(pid_t pid,
    const struct sched_attr *attr,
    unsigned int size,
    unsigned int flags);
```

4. EXPERIMENTAL RESULTS

In this section, we show some significant results obtained running a set of experiments on a PC based on an Intel Xeon L5640 CPU running at 2.26 GHz consisting of six cores and a 12 MB of cache.

4.1. Greedy tasks

As first experiment, we have used SCHED_DEADLINE to schedule one periodic task and two greedy tasks (i.e. tasks that never block and try to use the CPU at 100%). The periodic task executes for 1 ms every 4 ms. The greedy tasks have been assigned (1 ms, 6 ms) and (1 ms, 10 ms) reservations, respectively.

Figure 3 shows a segment of the schedule. The periodic task has the most strict timing requirements (relative deadline is 4 ms), therefore it always gets scheduled when it wakes up. Greedy 1 has higher priority than Greedy 2, and thus it pre-empts the latter during the first activation (remember that priorities are dynamic, the behaviour is thus relative to this particular timing window). Another thing to notice is that greedy tasks are throttled once they try to execute for more than the allowed runtime (red lines in the figure). The periodic task, instead, always blocks before exhausting its runtime per period, and thus it is never throttled.

This experiment shows that SCHED_DEADLINE is capable of providing timing confinement: it creates an effective isolation between the running tasks, so that greedy, buggy or misbehaving tasks cannot affect the execution of the other running tasks.

4.2. Synthetic real-time workloads

The next set of experiments has been designed to test the timing guarantees provided by SCHED_DEADLINE to real-time tasks, showing how SCHED_DEADLINE allows to properly schedule real-time applications. In these experiments, some sets of periodic real-time tasks have been randomly generated with taskgen [55] and executed by a user-level application (named `rt-app` [56]) either under the SCHED_DEADLINE, the SCHED_OTHER (i.e. Completely Fair Scheduler (CFS)), or the SCHED_FIFO (with proper priority assignments) scheduling policies. The number of missed deadlines has been measured as a function of the load $U = \sum \frac{C_i}{P_i}$, where C_i is the execution time of the i^{th} task and P_i is its activation period.

This experiment has been performed considering both partitioned scheduling and global scheduling. When using SCHED_DEADLINE, each task τ_i has been assigned a runtime Q_i slightly larger than its execution time C_i, and a server period T_i equal to the task period P_i; when using

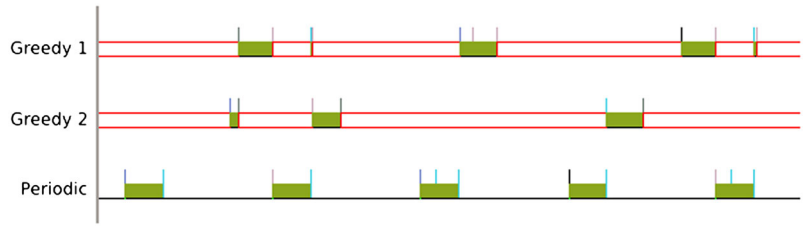

Figure 3. SCHED_DEADLINE serving a periodic task and two CPU hungry (greedy) tasks.

DOI: 10.1002/spe

SCHED_FIFO, priorities have been assigned using Rate Monotonic (RM) [27] in case of partitioned scheduling and DkC [57] in case of global scheduling.

In the partitioned scheduling case, tasks were statically bound to a CPU core (using the Linux `cpuset` mechanism), and the load on each core increased from 0.6 (60%) to 0.9 (90%). Notice that the $U = 1$ case (i.e. 100% CPU utilization) has been avoided in order to leave some spare time for the other tasks running in the system, so that the OS is not starved by SCHED_DEADLINE tasks. For each CPU load, 50 tasksets were randomly generated.

Table I shows the percentage of deadlines missed by each scheduling policy. For SCHED_DEADLINE, we used partitioned scheduling; for SCHED_FIFO, we used RM priority assignment. As it can be noticed, SCHED_DEADLINE is able to avoid any missed deadline (because the load on each core is smaller than 1, and because the scheduling parameters have been assigned in order to exploit the so-called *hard schedulability* property of the CBS). On the other hand, CFS performs pretty well, but is not able to avoid deadline misses (as it can be expected from a non real-time scheduling policy), while SCHED_FIFO avoids missed deadlines only for $U < 0.8$.

This experiment shows that if the taskset can be partitioned between the available cores (with a per-core load $U < 1$), then SCHED_DEADLINE is suitable for scheduling hard real-time tasks. SCHED_FIFO, on the other hand, can only handle tasksets with a smaller per-core load. It is also worth noticing that when using only CPU-intensive tasks (as in the previous experiments) the results obtained using SCHED_DEADLINE match the expectations from the real-time scheduling theory. Notice, however, that resource sharing (Section 3.3), I/O intensive tasks, or frequent invocation of system calls [58] can cause noticeable blocking times that affect the timing guarantees. Such blocking times must have an upper bound (e.g. using a proper resource sharing protocol as explained in Section 3.3 and a proper real-time kernel [12, 13]) and must be properly accounted for in the admission tests.

After testing partitioned scheduling, the experiment has been repeated by configuring SCHED_DEADLINE to do global EDF scheduling. In this case, the Linux `cpuset` mechanism has not been used so that tasks have been left free of migrating among all available CPU cores. First, some experiments have been performed using four of the six cores provided by the Xeon CPU, generating the tasksets with a total utilization ranging from $U = 3.4(340\%)$ to $U = 3.9(390\%)$. Again, the $U = 4$ case has not been considered in order to avoid starving the system. As for the previous experiment, 50 taskset per CPU load have been randomly generated.

Table II shows the percentage of deadlines missed by each scheduling policy. For SCHED_DEADLINE, we used global scheduling; for SCHED_FIFO, we used DkC priority assignment. In case of global scheduling, EDF is not able to guarantee that no deadline will be missed, thus

Table I. Percentage of missed deadlines for partitioned scheduling, as a function of the load U.

$U(\%)$	SCHED_DEADLINE (%)	SCHED_FIFO (%)	SCHED_OTHER (%)
60	0	0	0.58
70	0	0	1.87
80	0	0.003	6.03
90	0	0.38	10.20

Table II. Percentage of missed deadlines for global scheduling on four cores, as a function of the load U.

$U(\%)$	SCHED_DEADLINE (%)	SCHED_FIFO (%)	SCHED_OTHER (%)
340	0.008	0.010	3.75
350	0.012	0.436	6.17
360	0.030	0.095	6.92
370	0.117	0.231	8.52
380	0.186	0.702	10.62
390	0.615	2.297	14.15

DOI: 10.1002/spe

Table III. Percentage of missed deadlines for global scheduling on six cores, as a function of the load U.

U(%)	SCHED_DEADLINE (%)	SCHED_FIFO (%)	SCHED_OTHER (%)
500	0.027	0.001	3.303
510	0.023	0.002	4.310
520	0.051	0.011	4.992
530	0.099	0.023	6.046
540	0.138	0.230	7.093
550	0.239	0.271	8.097
560	0.289	0.380	9.977
570	0.351	0.640	11.554
580	0.618	1.380	15.384
590	1.295	2.535	19.774

SCHED_DEADLINE experiences some missed deadlines. The other schedulers, however, exhibit a higher percentage of missed deadlines (for large utilizations, the percentage of deadlines missed by DkC or CFS is more than twice the percentage experienced by the CBS).

Some additional experiments with global scheduling have been performed using all of the six cores of the Xeon CPU, and a CPU load ranging from $U = 5$ to $U = 5.9$. The results are presented in Table III, and show that in this case, SCHED_DEADLINE misses more deadlines than SCHED_FIFO (with DkC priorities) until the CPU load reaches $U = 5.4$. Then, for higher values of the CPU load, the dynamic priorities used by EDF provide better results than static priorities. This behaviour is because of the fact that SCHED_DEADLINE uses global EDF, without considering the task runtime when assigning dynamic priorities, while the DkC priority assignment is based on both periods and execution times (and it is well known that tasks with a high utilization are problematic for global scheduling algorithms, so their priority has to be increased). Again, as expected, CFS does not provide good real-time priorities.

Summing up, the experiments show that on multiprocessor (or multi-core) systems where the taskset cannot be statically partitioned between CPUs/cores and global scheduling must be used, SCHED_DEADLINE allows to reduce the number of missed deadlines when the CPU load is high, improving the performance of soft real-time tasks. Also, note that it has been proven that the global EDF algorithm (used by SCHED_DEADLINE in case of non-partitioned scheduling) always provide a bounded tardiness [59].

4.3. SCHED_DEADLINE on a real application

After showing how SCHED_DEADLINE helps in respecting the timing constraints of real-time tasks using some randomly generated synthetic workloads, the effectiveness of the new scheduling policy has been tested on a real application, showing how SCHED_DEADLINE allows to provide a reasonable quality to the application while controlling the amount of CPU time consumed by it. In particular, we have performed a set of tests using MPlayer[**], a simple yet widely used and powerful video player. Being single-threaded, it can be easily scheduled through a single reservation.

MPlayer has been modified to measure some important quality of service metrics when reproducing a video: the Inter-Frame Time (IFT) – defined as the difference between the display time of the current and the previous frame – and the Audio / Video desynchronisation (A/V Desynch) – defined as the difference between the Presentation TimeStamp (PTS) of the currently reproduced audio sample and the PTS of the currently reproduced video frame. Variations in the IFT can have a bad impact on the perceived video quality, because the video does not play smoothly, while large values of the A/V Desynch affect the quality of the reproduced media because audio and video do not appear synchronised (e.g. lip sync).

MPlayer has then been used to play an HD movie (with H.264 video and AAC+ audio), and scheduled by using SCHED_DEADLINE with a period equal to the expected IFT (1 / fps) and a

[**] http://www.mplayerhq.hu.

DOI: 10.1002/spe

runtime (maximum budged) ranging from 5 ms to 20 ms. Because the video frame rate is 23.976 fps, the expected IFT is $1\,000\,000/23.976 = 41708\mu s$. As expected, if the runtime was large enough, the IFT was stable around the expected value of $41708\mu s$. Decreasing the maximum budged, some jitter started to be visible in the IFT. Finally, for small values of Q_i, the IFT was out of control. Figure 4 shows the IFT measured for the first 600 frames with a value of the maximum runtime near to the one needed to decode without issues ($Q_i = 10$ ms) and a smaller value, which created issues and a non fluid playback ($Q_i = 7ms$).

Notice that a constant IFT near to the expected value can also be obtained by using SCHED_FIFO with a high real-time priority. However, SCHED_FIFO does not allow to control the percentage of CPU time consumed by MPlayer, while SCHED_DEADLINE allows to find a trade-off between the consumed CPU time and the perceived quality. For example, Figure 5 shows the impact of different Q_i values on the IFTs by plotting their Cumulative Distribution Function (CDF). Notice how increasing Q_i allows to make the CDF more similar to a step function (indicating that MPlayer has a probability near to 1 to play the video always at the correct rate), at the cost of dedicating more CPU time to MPlayer's execution. On the other hand, when scheduling MPlayer with SCHED_FIFO, the IFT CDF looked similar to the one obtained with $Q_i = 13$ms, but the CPU utilization of the task

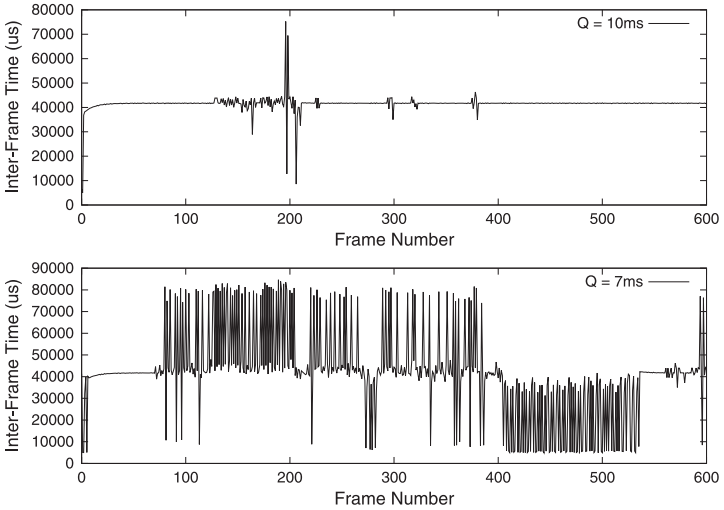

Figure 4. Inter-frame times for MPlayer scheduled with different SCHED_DEADLINE parameters.

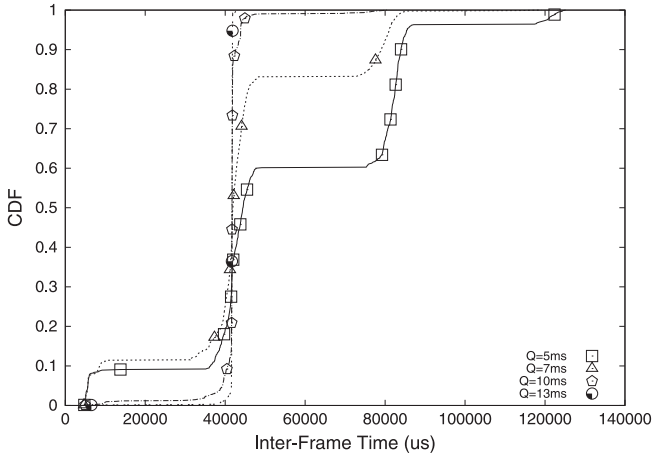

Figure 5. Cumulative distribution function (CDF) of the MPlayer's inter-frame times for different values of the SCHED_DEADLINE maximum runtime.

DOI: 10.1002/spe

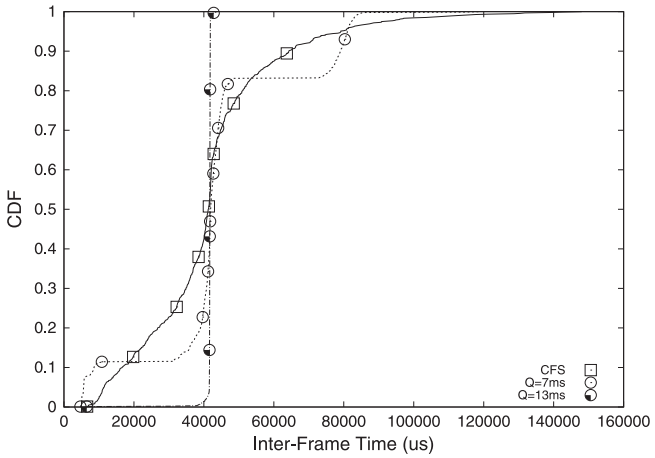

Figure 6. Cumulative distribution function (CDF) of the MPlayer's inter-frame times when SCHED_OTHER is used. CFS, completely fair scheduler.

ranged (according to the `pidstat` utility) from 5% to more than 35% and it was not possible to control it.

When, instead, MPlayer is scheduled with SCHED_OTHER (using the CFS – Completely Fair Scheduling – algorithm) the IFT CDF turned out to be highly dependent on the system load: with an unloaded system, the IFT CDF was similar to the one obtained with SCHED_FIFO (i.e. good performance); however, when some CPU-intensive activities were started in background, then the IFT CDF became much worse. For example, Figure 6 shows the IFT CDF obtained on a six-core CPU when running six instances of `ffmpeg` in background to transcode some videos. Obviously, when using SCHED_DEADLINE, MPlayer is not affected by the `ffmpeg`'s instances running in background (two of the CDFs obtained with SCHED_DEADLINE are also reported for comparison). The fact that CFS performs quite well in underload situations, but it does not perform as well as EDF-based scheduling for high loads, is well known in literature [60]. The CFS niceness parameter can be tuned to stabilise the IFTs when using SCHED_OTHER. However, this approach is far from providing the same amount of control of the SCHED_DEADLINE's runtime and period parameters. These parameters, in fact, allow to exactly set the percentage of execution time consumed by MPlayer, allowing to find better trade-offs between the quality of the video and the throughput – that is, number of frames transcoded per second – provided by the `ffmpeg` instances playing in background.

Summing up, SCHED_DEADLINE allows to both respect the temporal constraints of real applications (and not only synthetic benchmarks), and to find a proper trade-offs between QoS and CPU usage, while SCHED_FIFO and SCHED_OTHER do not allow to find such trade-offs.

Similarly, to the experiments performed in [40], Figure 7 displays the A/V Desynch experienced for different values of Q_i. The results show again that SCHED_DEADLINE can be effectively used to control the quality perceived by a user and to guarantee the proper behaviour of time-sensitive applications. Again, SCHED_OTHER with a small CPU load and SCHED_FIFO result in an A/V Desynch close to 0. SCHED_OTHER with a high CPU load results in an A/V Desynch comparable with the one obtained with SCHED_DEADLINE and a runtime of $9ms$; playing with the niceness parameter allows to reduce the A/V Desynch, but does not provide as much control as SCHED_DEADLINE.

Summing up, SCHED_DEADLINE provides a good amount of control over the real-time performance of real applications, because it allows to better specify the applications' parameters and requirements: because the user can communicate to the scheduler some temporal constraints to be respected (in the form of a period T_i and a runtime Q_i), the scheduler can do a better job in trying to respect these constraints.

DOI: 10.1002/spe

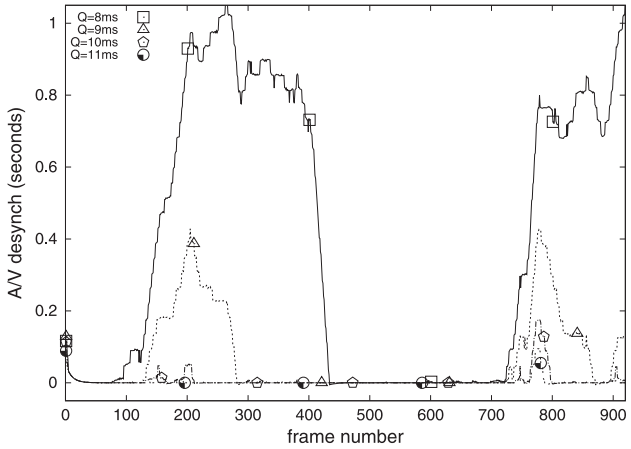

Figure 7. Audio/video dsynchronisation (A/V Desynch) time for MPlayer scheduled by SCHED_DEADLINE with different values of the maximum runtime.

4.4. Using SCHED_DEADLINE to control the application throughput

The previous experiments showed how SCHED_DEADLINE allows to respect the deadlines of real-time applications and to properly serve 'legacy' time-sensitive applications that have not been specifically designed to be used with SCHED_DEADLINE (controlling the trade-off between their quality and CPU usage). In other words, these experiments focused on the *timing guarantees* that can be provided based on the temporal isolation property of the CBS.

However, the temporal isolation property also allows to enforce timing confinement to control the throughput of non real-time (or time-sensitive) applications – that is, SCHED_DEADLINE can be used even in situations where real-time performance is not strictly required. This last set of experiments has been performed to illustrate this possibility.

The application used in these experiments is Kernel-based virtual machine (KVM) [61], used to run a Virtual Router (VR) [62]. This VR is implemented by executing a software router (a Linux-based OS running Quagga[††] as a routing daemon and using the Linux kernel as a data plane) in a KVM virtual machine. KVM creates a user-space thread (the vCPU thread) for executing the software router and uses a kernel thread (the vhost-net kernel thread) to move packets between the virtualised software router and the physical network cards. Because both of these threads tend to consume a lot of CPU time (when the rate of packets to be routed is too high), SCHED_DEADLINE can be used to control their CPU usage, seeing how the fraction of CPU time reserved to these threads can affect the router performance. Notice that the two threads are not pure 'CPU-hungry' tasks (consuming CPU time in user-space without interacting with the kernel), but tend to stress the kernel very much (invoking a good amount of system calls and kernel functions, causing context switches, and even invoking hypervisor calls) and sometimes behave as self-suspending tasks. Hence, an additional goal for this experiment is to show that SCHED_DEADLINE can also handle non-trivial loads composed by self-suspending tasks that stress the kernel.

In this experiment, performed by using VRKit [63], the VR has been fed with small (64 bytes) UDP packets, while scheduling the vCPU thread and the vhost-net kernel thread with SCHED_DEADLINE and reserving different percentages of CPU time to them (remember that the percentage of CPU time reserved to a task scheduled by SCHED_DEADLINE with runtime Q_i and period T_i is equal to $\frac{Q_i}{T_i}$ in percentage).

Figures 8 and 9 plot the rate of routed packets as a function of Q_i/T_i (in percentage). SCHED_DEADLINE has been used to schedule both the vCPU thread and the vhost-net kernel thread. The experiment has been performed for several input packet rates, but only some interesting rates have been reported in the figures. The lower line corresponds to an underloaded VR: increasing

[††]http://www.nongnu.org/quagga.

DOI: 10.1002/spe

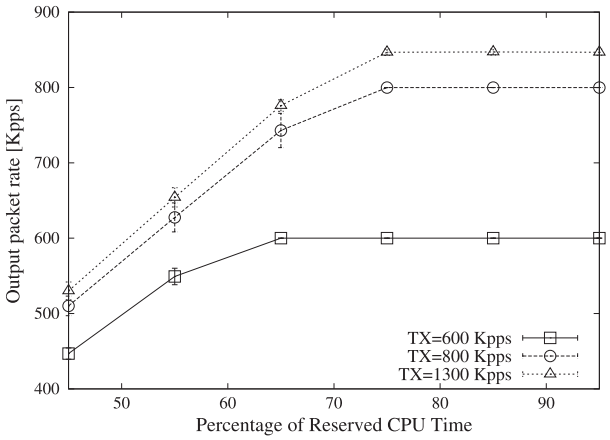

Figure 8. Throughput of a KVM-based virtual router as a function of the percentage of CPU time reserved to the vCPU thread.

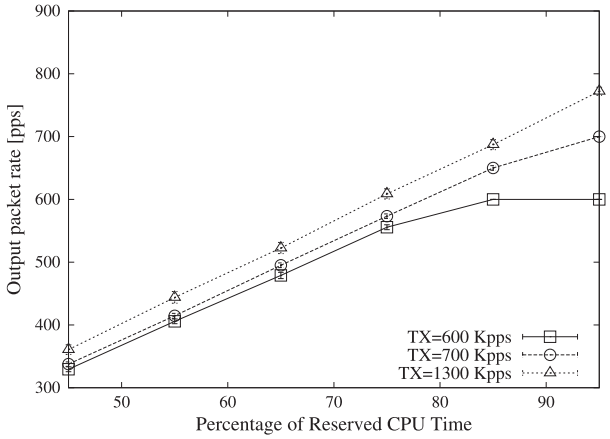

Figure 9. Throughput of a Kernel virtual machine-based virtual router as a function of the percentage of CPU time reserved to the vhost-net kernel thread.

Q_i/T_i, the routed packets rate reaches the input rate, and the line becomes flat (i.e. reserving more CPU time to KVM cannot improve the performance). The medium line corresponds to a system near to overload. The higher line corresponds to an overloaded VR (i.e. maximum possible input packet rate): in this case, even when the KVM thread is reserved 95% of the CPU time, the routed packets rate cannot reach the input rate.

The interesting thing to be noticed is that the VR performance (the routed packets rate) increases almost linearly with the ratio Q_i/T_i (fraction of CPU time reserved to the vCPU or vhost-net thread). This shows that SCHED_DEADLINE allows to easily control the performance (even non real-time performance) of the applications. The reason is that SCHED_DEADLINE allows to exactly control the fraction of CPU time used by the vhost-net kernel thread or by the vcpu thread. The other schedulers offered by Linux do not provide a similar level of control.CFS, for example, allows to assign a weight to each thread, and to share the CPU time between threads proportionally to such weights, but cannot guarantee that a thread receives exactly the desired fraction of CPU time (this would require to avoid all tasks' migrations, or to recompute proper task weights whenever a task is migrated). Finally, similar results can be obtained by scheduling the vhost-net kernel thread and the vcpu thread with high real-time priorities, and using the Linux 'Traffic Control' (TC) mechanisms to limit the VR throughput. However, this technique does not allow to precisely control the fraction of CPU time consumed by the threads; hence, it is not possible to estimate their interference on other

DOI: 10.1002/spe

tasks. For example, if some real-time tasks are scheduled on the same CPU cores as vhost-net and vcpu, it is not possible to provide real-time guarantees to such threads. Using SCHED_DEADLINE, instead, this is possible.

5. CONCLUSIONS

In this paper, we presented SCHED_DEADLINE, the real-time scheduler recently merged into the official Linux kernel. We have described the scheduling paradigm (i.e. Resource Reservation), the implemented algorithm (i.e. CBS), the internal design and the API available to the software developers. The experimental results on the real hardware showed that SCHED_DEADLINE can effectively provide temporal isolation among the running tasks and reduce the number of deadline misses experienced by the real-time tasks.

The future work will be focused on the following:

- Investigation of more sophisticated Resource Reservation algorithms (e.g. IRIS [25]);
- Investigation of the high overhead shown by SCHED_DEADLINE in some circumstances [41];
- Development of power-aware real-time algorithms (e.g. GRUB-PA [32]) to reduce power consumption still meeting real-time constraints;
- Development of M-BWI [54] as replacement of deadline inheritance, to solve the issues described in Section 3.3;
- Access control for not privileged users;
- Bandwidth management through the cgroups interface [29].

ACKNOWLEDGEMENTS

The authors would like to thank Peter Zijlstra and Ingo Molnar for the trust and the precious technical guidance. Further thanks to Michael Trimarchi and Fabio Checconi for technical suggestions and testing. Special thanks to Johan Eker, Paolo Gai, Tommaso Cucinotta and Giuseppe Lipari for having supported the project throughout these years.

ACTORS FP7 project, JUNIPER FP7 project:216586, 318763.

REFERENCES

1. Lipari G, Scordino C. Linux and real-time: current approaches and future opportunities. *IEEE International Congress ANIPLA '06*, Rome, Italy, 2006.
2. Dozio L, Mantegazza P. Real-time distributed control systems using RTAI. *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, Hakodate, Hokkaido, Japan, 2003; 11–18.
3. Yodaiken V. The RTLinux manifesto. *Proceedings of the 5th Linux Expo*, Raleigh, North Carolina, 1999.
4. Wind River acquires hard real-time Linux technology from FSMLabs, 2007. Press release, Available at: http://www.windriver.com/uk/press/pr.html?ID=4301 [last accessed 28 May 2015].
5. RTAI, the Real-Time Application Interface for Linux. Available at: https://www.rtai.org [last accessed 28 May 2015].
6. Yaghmour Karim. Adaptive domain environment for operating systems. *Opersys inc*, 2001.
7. Xenomai, Real-time Framework for Linux. Available at: http://www.xenomai.org [last accessed 28 May 2015].
8. Barbalace A, Luchetta A, Manduchi G, Moro M, Soppelsa A, Taliercio C. Performance comparison of VxWorks, Linux, RTAI and Xenomai in a hard real-time application. In *2007 15th ieee-npss Real-time conference*. IEEE: Batavia, IL, USA, 2007; 1–5.
9. Morton A. *Low latency patches*. Web archive Available at: https://web.archive.org/web/20080306131124/http://www.zipworld.com.au/~akpm/linux/schedlat.html [last accessed 28 May 2015].
10. Love R. *Linux Kernel Development* (3rd edition). Addison-Wesley Professional: Boston, USA, 2010.
11. Williams C. Linux scheduler latency, 2002. Red Hat Inc., WhitePaper Available at: http://www.eetimes.com/document.asp?doc_id=1200916 [last accessed 28 May 2015].
12. PREEMPT_RT patch. Available at: https://rt.wiki.kernel.org [last accessed 28 May 2015].
13. Rostedt S. Internals of the rt patch. *Proceedings of the Linux Symposium*, Ottawa, Canada, 2007; 161–172.
14. Yan Y, Cosgrove S, Anand V, Kulkarni A, Konduri SH, Ko SY, Ziarek L. Real-time android with rtdroid. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. ACM: Bretton Woods, NH, USA, 2014; 273–286.
15. Valente P, Checconi F. High throughput disk scheduling with fair bandwidth distribution. *IEEE Transactions on Computers* 2010; **59**(9):1172–1186.

DOI: 10.1002/spe

16. Valente P. Providing near-optimal fair-queueing guarantees at round-robin amortized cost. *Proceedings of the 22nd International Conference on Computer Communication and Networks (ICCCN)*, Nassau, Bahamas, 2013.
17. Checconi F, Valente P, Rizzo L. QFQ: Efficient packet scheduling with tight bandwidth distribution guarantees. *IEEE/ACM Transactions on Networking* 2012; **21**(3):802–816.
18. Abeni L, Lipari G. Implementing resource reservations in Linux. *Real-Time Linux Workshop*, Boston (MA), 2002.
19. Strosnider JK, Lehoczky JP, Sha L. The deferrable server algorithm for enhanced aperiodic responsiveness in hard-real-time environments. *IEEE Transactions on Computers* 1995; **44**(1):73–91.
20. Joseph M, Pandya P. Finding response times in a real-time system. *The Computer Journal* 1986; **29**(5):390.
21. Mercer CW, Rajkumar R, Tokuda H. Applying hard real-time technology to multimedia systems. *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.
22. Abeni L, Buttazzo G. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. IEEE: Madrid, Spain, 1998; 4–13.
23. Lipari G, Baruah SK. Greedy reclamation of unused bandwidth in constant bandwidth servers. *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, 2000; 193–200.
24. Scordino C. Dynamic voltage scaling for energy-constrained real-time systems. *Ph.D. Thesis*, University of Pisa, 2007.
25. Marzario L, Lipari G, Balbastre P, Crespo A. IRIS: a new reclaiming algorithm for server-based real-time systems. *Real-Time Application Symposium (RTAS)*, Toronto, Canada, 2004; 211–218.
26. Oikawa S, Rajkumar R. Portable RK: a portable resource kernel for guaranteed and enforced timing behavior. *Proceedings of 5th Real-Time Technology and Applications Symposium*, Vancouver, British Columbia, 1999; 111–120.
27. Liu CL, Layland JW. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery* 1973; **20**(1):46–61.
28. Faggioli D, Mancina A, Checconi F, Lipari G. Design and implementation of a POSIX compliant sporadic server. *Proceedings of the 10th Real-Time Linux Workshop (RTLWS)*, Mexico, 2008.
29. *Linux control groups*, 2007. Available at: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt [last accessed 28 May 2015].
30. Stanovich M, Baker TP, Wang AI, Harbour MG. Defects of the POSIX sporadic server and how to correct them. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE: Stockholm, Sweden, 2010; 35–45.
31. Ripoll I, Pisa P, Abeni L, Gai P, Lanusse A, Saez S, Privat B. Wp1 - rtos state of the art analysis: deliverable d1.1 - rtos analysis. Technical report, OCERA, 2002.
32. Scordino C, Lipari G. A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks. *IEEE Transactions on Computers* 2006; **55**(12):1509–1522.
33. Paolopoli L, Cucinotta T, Marzario L, Lipari G. AQuoSA-adaptive quality of service architecture. *Software: Practice and Experience* 2009; **39**(1):1–31.
34. Calandrino JM, Leontyev H, Block A, Devi UC, Anderson JH. LITMUS^{RT}: a testbed for empirically comparing real-time multiprocessor schedulers. *Proceedings of the 27th IEEE Real-Time Systems Symposium*, Rio De Janeiro, Brazil, 2006; 111–123.
35. Bastoni A, Brandenburg BB, Anderson JH. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *2010 IEEE 31st Real-Time Systems Symposium (RTSS)*. IEEE: San Diego, California, USA, 2010; 14–24.
36. Asberg M, Nolte T, Kato S, Rajkumar R. Exsched: an external cpu scheduler framework for real-time systems. In *2012 IEEE 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE: Seoul, Korea, 2012; 240–249.
37. Libenzi D. SCHED_SOFTRR Policy. Available at: http://xmailserver.org/linux-patches/softrr.html [last accessed 28 May 2015].
38. Kolivas C. Isochronous class for unprivileged soft RT scheduling. Available at: http://ck.kolivas.org/patches/ [last accessed 28 May 2015].
39. Checconi F, Cucinotta T, Faggioli D, Lipari G. Hierarchical multiprocessor CPU reservations for the Linux Kernel. *Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, Dublin, Ireland, 2009; 15–22.
40. Kato S, Ishikawa Y, Rajkumar RR. Cpu scheduling and memory management for interactive real-time applications. *Real-Time Systems* 2011; **47**(5):454–488.
41. Cerqueira F, Vanga M, Brandenburg BB. Scaling global scheduling with message passing. *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2014)*, Berlin, Germany, 2014; 263–274.
42. Faggioli D, Checconi F, Trimarchi M, Scordino C. An EDF scheduling class for the Linux kernel. *11th Real-Time Linux Workshop (RTLWS)*, Dresden, Germany, 2009.
43. SCHED_EDF scheduling class. Available at: https://lwn.net/Articles/353797/ [last accessed 28 May 2015].
44. Request to change the name from SCHED_EDF to SCHED_DEADLINE. Available at: http://thread.gmane.org/gmane.linux.kernel/892838 [last accessed 28 May 2015].
45. Sha L, Rajkumar R, Lehoczky JP. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transaction on Computers* 1990; **39**(9):1175–1185.
46. Spuri M, Buttazzo GC. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems* 1996; **10**(2):179–210.

DOI: 10.1002/spe

47. Ghazalie TM, Baker TP. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time System* 1995; **9**:31–67.
48. Spuri M, Buttazzo G. Efficient aperiodic service under earliest deadline scheduling. *Proceedings of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, 1994.
49. Lelli J, Faggioli D, Cucinotta T, Lipari G. An experimental comparison of different real-time schedulers on multicore systems. *Journal of Systems and Software* 2012; **85**(10):2405–2416.
50. Lelli J, Lipari G, Faggioli D, Cucinotta T. An efficient and scalable implementation of global EDF in Linux. *Proceedings of the 7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2011)*, Porto, Portugal, 2011; 6–15.
51. Gujarati A, Cerqueira F, Brandenburg BB. Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities. In *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE: Paris, France, 2013; 69–79.
52. Jansen PG, Mullender SJ, Havinga PJM, Scholten H. Lightweight edf scheduling with deadline inheritance, 2003.
53. Lipari G, Lamastra G, Abeni L. Task synchronization in reservation-based real-time systems. *IEEE Transactions on Computers* 2004; **53**(12):1591–1601.
54. Faggioli D, Lipari G, Cucinotta T. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems* 2012; **48**(6):789–825.
55. Emberson P, Stafford R, Davis RI. Techniques for the synthesis of multiprocessor tasksets. *Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS 2010)*, Brussels, Belgium, 2010; 6–11.
56. rt-app scheduling tool. Available at: https://github.com/scheduler-tools/rt-app [last accessed 28 May 2015].
57. Davis RI, Burns A. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, Washington, DC, 2009; 348–409.
58. Abeni L, Goel A, Krasic C, Snow J, Walpole J. A measurement-based analysis of the real-time performance of linux. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*. IEEE: San Jose, California, 2002; 133–142.
59. Devi UC, Anderson JH. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Systems* 2008; **38**(2):133–189.
60. Kenna CJ, Herman JL, Brandenburg BB, Mills AF, Anderson JH. Soft real-time on multiprocessors: are analysis-based schedulers really worth it? *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, Vienna, 2011; 93–103.
61. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. KVM: the Linux Virtual Machine Monitor. *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2007; 225–230.
62. Abeni L, Kiraly C, Li N, Bianco A. Tuning KVM to enhance virtual routing performance. In *Proceedings of the IEEE ICC'2013*. IEEE: Budapest, Hungary, 2013; 2396–2401.
63. Abeni L, Kiraly C. Running repeatable and controlled virtual routing experiments. *Software: Practice and Experience* 2015; **45**:455–471.

DOI: 10.1002/spe