Energy-Aware Real-Time Scheduling in the Linux Kernel

Claudio Scordino
Evidence Srl
San Giuliano Terme, Pisa, Italy
claudio@evidence.eu.com

Luca Abeni
Scuola Superiore S. Anna
Pisa, Italy
luca.abeni@santannapisa.it

Juri Lelli
Red Hat, Inc.
Raleigh, NC
juri.lelli@redhat.com

ABSTRACT

The recent changes made in the Linux kernel aim at achieving better energy efficiency through a tighter integration between the CPU scheduler and the frequency-scaling subsystem. However, in the current implementation, the frequency scaling mechanism is used only when there are no real-time tasks in execution. This paper shows how the deadline scheduler and the cpufreq subsystem can be extended to relax this constraint and implement an energy-aware real-time scheduling algorithm. In particular, we describe the design issues encountered when implementing the GRUB-PA algorithm on a real operating system like Linux. A set of experimental results on a multi-core ARM platform validate the effectiveness of the proposed implementation.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; **Real-time operating systems**;

KEYWORDS

DVFS, Power-Aware Scheduling, Real-Time Scheduling, Linux

ACM Reference Format:
Claudio Scordino, Luca Abeni, and Juri Lelli. 2018. Energy-Aware Real-Time Scheduling in the Linux Kernel. In *SAC 2018: SAC 2018: Symposium on Applied Computing , April 9–13, 2018, Pau, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3167132.3167198

1 INTRODUCTION

During the recent years, the ICT industry has faced a growing pressure for increasing the processing capabilities of mobile devices (like smartphones or IoT nodes) and, at the same time, extending (or, at least, not reducing) their autonomy. The battery technology on this kind of devices, in fact, has evolved at a too slow pace for being capable of satisfying the processing needs posed by next-generation applications. To make things harder, on these devices there is often the additional requirement of avoiding the mechanical cooling systems typically used for dissipating the heat produced by a powerful processing unit (because they could easily break and, in any case, would increase the overall size and energy consumption).

Dynamic Voltage and Frequency Scaling (DVFS) is an effective technique for reducing the power-consumption of CPUs by dynamically lowering their frequency and voltage. An energy-aware scheduler running on a General-Purpose Operating System (GPOS) can exploit this mechanism to select both the task to be executed and the CPU's Operating Performance Point (OPP), aiming at reducing the overall energy consumption.

The DVFS technique can be effectively used also for real-time tasks, with the objective of setting the CPU at the lowest frequency that allows to respect the real-time constraints (e.g., a specified amount of computation before task's deadline). Several energy-aware scheduling algorithms have been proposed in the real-time literature during the years. Pillai and Shin [16] proposed the RT-DVS family of algorithms to exploit the slack time. Aydin et al. [7] proposed the DRA algorithms for reclaiming the spare time on Earliest Deadline First (EDF). A power-aware algorithm for EDF scheduling has been proposed by Zhu and Mueller [24] as well. Similar techniques have been proposed by Saewong and Rajkumar [21] in the context of fixed priority scheduling. All these techniques assume hard real-time *periodic* task sets and therefore do not fit the scenario of GPOS kernels like Linux.

Some algorithms have been proposed for soft real-time tasks too. For example, Lorch and Smith addressed variable voltage scheduling of tasks with soft deadlines in [14]. Pouwelse et al. [17, 18] presented a study of power consumption and power-aware scheduling applied to multimedia streaming. Kumar et al. [11] proposed a prediction mechanism for fixed-priority scheduling of soft periodic tasks. However, these techniques are based on heuristics and cannot provide guarantees to hard real-time tasks. Qadi et al. [19] presented the DVSST algorithm that reclaims the unused bandwidth of sporadic hard real-time tasks.

The GRUB-PA algorithm [23] supports periodic, sporadic and aperiodic tasks and has been shown to outperform most of the mentioned algorithms. Moreover, it can be used to schedule both hard and soft real-time tasks. Hence, it looks like the best candidate to implement DVFS in general purpose kernels such as Linux. This paper presents and evaluates the implementation of a new DVFS mechanism for the Linux kernel based on such algorithm.

The rest of the paper is organized as follows. Section 2 illustrates which of the existing Linux components our implementation relies on. Section 3 outlines the general approach and its theoretical foundations. Section 4 describes the implementation details. Section 5 provides a set of experimental results on a real embedded hardware. Finally, Section 6 draws the conclusions.

2 BACKGROUND

This section illustrates the "building blocks" that we have used for implementing the proposed energy-aware real-time scheduler.

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167198

2.1 Deadline scheduling

Since release 2.6.23, Linux has a modular scheduling framework consisting of a main core 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 [12] is a scheduling policy suitable for real-time applications, available by default since the 3.14 kernel release. It implements the *Constant Bandwidth Server* (CBS) [2] CPU reservation algorithm over an *Earliest Deadline First* (EDF) [13] scheduler. Each real-time task is assigned a *"reservation"* (Q_i, T_i), meaning that the task needs to execute at least for the *"runtime"* Q_i[1] every *period* of time T_i. The ratio Q_i/T_i is called "utilization" of the task. Thanks to the EDF optimality, if the sum of the utilizations of the tasks on a CPU does not exceed 100% (i.e. $\sum \frac{Q_i}{T_i} \leq 1$), then the tasks are guaranteed to respect their execution constraints.

If a task tries to execute for more than Q_i in a period T_i, then it gets *throttled* (i.e., not selected for execution) until the end of the current reservation period. This effect is achieved by using three mechanisms:

- accounting: each task is associated to a *current runtime*, that is decreased when the task executes. In particular, if the task executes for a time δ, its current runtime is decreased by δ.
- enforcement: when the current runtime of a task arrives to 0, the task is throttled and cannot be scheduled until the current runtime is replenished.
- replenishment: at the end of a reservation period (when the time becomes equal to the scheduling deadline of the task), the current runtime of the task is replenished to the maximum value Q_i (and the scheduling deadline is postponed by T_i).

In this way, each task is constrained to not use more than its reserved CPU share — i.e., a maximum of Q_i every T_i units of time. This behavior (known as "hard reservation" [20]) has been designed to avoid the "deadline aging" phenomenon [5, 22], where a task consumes its future reservation due to other real-time tasks not ready to run. Moreover, the hard reservation behaviour avoids the starvation of lower priority tasks due to misbehaving real-time tasks (i.e., *isolation* property). On the other hand, however, it makes the scheduler not work conserving, because a real-time task might be throttled even in case of idle system.

2.2 Reclaiming

Since the recent 4.13 kernel release, a new "CPU reclaiming" feature, based on the Greedy Reclamation of Unused Bandwidth (GRUB) algorithm [3, 10], has been added to SCHED_DEADLINE. By introducing the reclaiming feature, GRUB solves the work conserving issue previously illustrated. This feature is obtained by keeping track of the CPU utilization U^{act} of the tasks that are active on each CPU core, and by using this information to modify the accounting rule. Considering the task τ_i served by a reservation (Q_i, P_i), the active utilization is immediately increased by $U_i = Q_i/P_i$ when τ_i wakes up (i.e., it is added to the runqueue of the scheduler). When τ_i blocks, however, the active utilization cannot be immediately decreased, otherwise this could break the real-time guarantees (e.g., if τ_i unblocks immediately later and the bandwidth has been already reclaimed by another task). Therefore, when τ_i blocks, a timer is armed to fire at the so-called "0-lag time", when the task's utilization can be safely removed from U^{act}. If τ_i unblocks before the 0-lag time, then the timer is cancelled. Figure 1 shows the state transitions for a GRUB task.

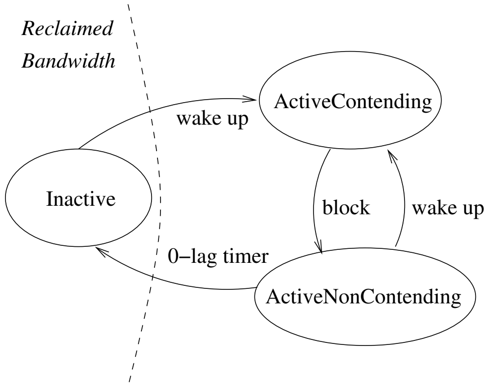

Figure 1: Task state diagram in GRUB.

The modified accounting rule defined by GRUB can be selected per-task, by using the new `SCHED_FLAG_RECLAIM` flag introduced in the user-space API. If set, this flag allows a task to explicitly reclaim some further bandwidth (if any) unused by the other real-time tasks. At the same time, to avoid starvation of lower priority tasks, the scheduler maintains a (configurable) margin of spare CPU bandwidth for running non real-time tasks on fully loaded systems.

The GRUB algorithm can be divided into two different parts: a set of rules for identifying the reclaimable bandwidth, and a set of rules for exploiting such bandwidth. The second part is where the GRUB and GRUB-PA algorithms differ. The original GRUB algorithm reassigns the reclaimed bandwidth to let the active real-time tasks execute for a longer time. In the GRUB-PA algorithm [23], instead, the reclaimed CPU time is used to slow down the processor (i.e., to lower the CPU frequency). The real-time tasks execute for a longer time, but at a slower speed. The net effect is a reduction of the CPU energy consumption still respecting the real-time guarantees.

The specification of the two algorithms is very similar, as GRUB-PA only adds a couple of extensions:

- it considers the processor speed when performing the runtime accounting;
- it sets the processor speed equal to U^{act}.

The interested readers can refer to the original papers for an in-depth description of the algorithms.

2.3 Frequency scaling

Cpufreq is the Linux subsystem in charge of adjusting CPU's voltage and frequency. It contains a set of "governors", each adopting a different power management strategy (in other words, the cpufreq

[1] Notice that the Linux "runtime" is often called "budget" in the real-time literature.

core implements the frequency scaling mechanism, while the various governors implement different policies).

The *ondemand* and *conservative* governors aim at dynamically adjusting the CPU frequency based on the system load. However, they have two issues: their goal is not formally specified (to correctly schedule real-time tasks, instead, it is important to formalize the invariants that the DVFS mechanism has to respect) and they have poor performance due to a coarse integration with the CPU scheduler. To address the second issue, the Linux kernel introduced the *schedutil* governor (since version 4.7), bridging the cpufreq subsystem with the CPU scheduler. Load estimation for non real-time tasks is achieved through the scheduler's Per-Entity Load Tracking (PELT) mechanism, which gives more importance to recent load contributions by using a geometric series.

Unfortunately, due to the first issue mentioned above, the current schedutil governor performs DVFS only for the SCHED_ OTHER tasks (managed by the CFS scheduling class) while real-time tasks (SCHED_FIFO, SCHED_RR and SCHED_DEADLINE) are always run at the highest CPU frequency.

3 APPROACH

In this paper, the schedutil governor has been improved to allow some energy saving when real-time tasks are scheduled. To do this, first of all it is important to precisely define the goal of the DVFS mechanism. This requires some basic definitions about real-time tasks: a real-time task τ_i can be seen as a sequence of *jobs* (or *instances*) $J_{i,j}$, with each job $J_{i,j}$ activating at time $r_{i,j}$ and executing for a time $c_{i,j}$ before the task blocks waiting for the next job. If a job $J_{i,j}$ finishes before time $r_{i,j} + D_i$ (with D_i relative deadline of the task), then its deadline is respected.

While setting the CPU at the maximum frequency is reasonable for SCHED_FIFO and SCHED_RR tasks (because their CPU requirements are not known in advance), a smarter approach can be taken when scheduling SCHED_DEADLINE tasks, whose CPU demand is specified explicitly in terms of runtime and period. In particular, the active utilization U^{act}, already tracked by the GRUB implementation in the Linux kernel, represents the fraction of CPU time used by the SCHED_DEADLINE tasks currently running on a CPU core.

The goal of scheduler and DVFS mechanism is to respect the tasks' deadlines. If a real-time task τ_i is scheduled by SCHED_DEADLINE with $Q_i \geq \max_j\{c_{i,j}\}$, $T_i \leq D_i \cup T_i \leq \min_j\{r_{i,j+1} - r_{i,j}\}$ and some admission test is respected[2], then the task is guaranteed to respect all of its deadlines. Hence, the new DVFS mechanism must be defined not to break this guarantee.

Considering a single CPU core, if its frequency is set to the maximum value f^{max}, then the core will not execute SCHED_DEADLINE tasks for a fraction $(1 - U^{act})$ of the time. Slowing down the CPU frequency, the jobs' execution times $c_{i,j}$ will increase proportionally; hence all the runtimes Q_i have to be increased proportionally in order not to miss any deadline. As a consequence, the amount of time not used by SCHED_DEADLINE tasks (which can be seen as an "idle time"[3]) will decrease. If the CPU frequency f is higher than $f^{max} \cdot U^{act}$, then it is possible to guarantee that each job of each SCHED_DEADLINE task will be able to finish before its deadline (that is, it will be able of receiving an amount of CPU time equal to $\frac{c_{i,j} \cdot f^{max}}{f}$ before the end of the reservation period). Hence, it is possible to save some energy by setting the CPU frequency to $f = f^{max} \cdot U^{act}$, as requested by GRUB-PA.

Again, remember that since lowering the CPU frequency slows down the performance, the tasks will need more time to do their work (in other words, job $J_{i,j}$ will execute for an amount of time $\frac{c_{i,j} \cdot f^{max}}{f}$ instead of $c_{i,j}$). For this reason, the runtimes have to be rescaled accordingly. This result can be achieved by modifying the accounting rule: when a task executes for a time δ, its current runtime is not decreased by δ, but by $\delta \cdot f^{max}/f$ (where f is the current frequency). If $f = f^{max} \cdot U^{act}$ then the current runtime is decreased by $\delta \cdot U^{act}$, as done by GRUB (and by the new reclaiming mechanism set by SCHED_FLAG_RECLAIM).

Note that, in line with the other DVFS mechanisms available in Linux, we have assumed the speed of the executed task to be proportional to the CPU frequency. Moreover, it is important to point out that changing the frequency of the cores affects their processing speed, but not the latency of the memory accesses. More complex models can be elaborated to take into account both the processing and the memory access speeds.

Since real CPUs do not allow to set their operating frequencies to arbitrary values (but permit to select only a limited number of discrete values), the cpufreq subsystem selects the minimum possible frequency that is higher than $f^{max} \cdot U^{act}$. Additionally, it automatically discards requests of setting a CPU frequency equal to the one already in use.

Summing up, the DVFS mechanism proposed in this paper, inspired by GRUB-PA, scales the CPU frequency based on the active utilization U^{act}. This result has been obtained by modifying the schedutil governor to use U^{act} as an estimation of the CPU load: when considering only SCHED_DEADLINE tasks, the resulting frequency scaling is identical to the one obtained by using GRUB-PA.

4 DESIGN AND IMPLEMENTATION

While implementing the schedutil modifications described in Section 3, the theoretical GRUB-PA algorithm has been slightly modified to address some implementation issues as described in this section.

4.1 Tracking the Utilization

Since the Linux kernel supports multi-processor (and multi-core) systems, the reclaiming mechanism implemented on top of SCHED_ DEADLINE is based on the M-GRUB [4] algorithm, which extends the original GRUB algorithm to deal with multiple CPUs. Originally, GRUB [10] and GRUB-PA [23], in fact, were designed for single-core systems, and thus used a single variable to keep track of the overall active utilization (given by all ActiveContending and ActiveNonContending tasks). The amount of reclaimable CPU time was computed based on the difference between the maximum CPU utilization (i.e., 100%) and such variable. The M-GRUB implementation that has been recently merged in the Linux kernel, instead, tracks the active utilization per-runqueue (that is, per CPU core).

[2] The admission test for single-processor systems is $\sum_i \frac{Q_i}{T_i} \leq 1$. For multi-processor systems it is more complex — for example see Section 4.2 of [4].

[3] Notice that the term "idle" is slightly incorrect here, because the CPU is not idle but it is simply not executing SCHED_DEADLINE tasks.

Moreover, M-GRUB needs to explicitly track the "inactive utilization" (defined as the difference between the utilization of all the tasks assigned to a CPU core and all the "active" tasks on the core). Hence, the kernel keeps track of two different utilizations per runqueue: the original "active" utilization and an additional "total" utilization (taking into account also any non-active task). The inactive utilization is computed as the difference between these two values.

One of the first design choices when implementing GRUB-PA has therefore concerned which of these two utilizations to use for frequency scaling. Suppose that the jobs of a real-time task τ_i with an utilization $U_i = Q_i/T_i = 70\%$ finish after using a much lower runtime. Setting the CPU frequency based on the total runqueue utilization (i.e., 70% in the previous example) is a more conservative approach, that has the drawback of a poor energy efficiency. With such an approach, in fact, a real-time task contributes to the CPU frequency even when blocked. On the other hand, it has the benefit of a lower number of frequency switches (because the frequency is changed only at task creation and destruction). Considering that the WCET is often much higher than the average execution time, and aiming at reaching a better energy performance, we have rather preferred a more aggressive approach. We have followed the GRUB-PA algorithm more strictly by relying on the active bandwidth. This approach has the advantage of further reducing the CPU frequency whenever a blocked task enters the inactive state.

4.2 Runtime Accounting

The frequency set by the schedutil governor has of course to take into account also the processing needs of the other scheduling classes. This has been easily achieved by extending the existing data structures to keep track of the load contributions of the various scheduling classes.

In the original GRUB-PA algorithm, the accounting operations (i.e. decreasing the remaining runtime of the executing task) were performed assuming that the CPU frequency had been set only based on the active utilization of `SCHED_DEADLINE` tasks. Since the CPU frequency can be different from $f^{max} \cdot U^{act}$ (due to the load contributions of the other scheduling classes), the accounting must be performed considering the actual frequency, and not the value of U^{act} (as originally done by GRUB and GRUB-PA). In this way, the current runtime of the task is decreased based on how much processing power (i.e., CPU frequency) has been actually given to it.

A more critical issue concerns the other scheduling classes being free of updating their own load contributions (thus affecting the CPU frequency) without informing the deadline scheduler of such changes. This means that when performing the runtime accounting, the real-time task could have been executed at a frequency different than the one currently used. We had three possible options to deal with these asynchronous frequency changes:

(1) Add a notification mechanism to inform the deadline scheduler about every change of the CPU frequency made by the other scheduling classes. Despite the precise accounting, however, this approach would have introduced a large amount of unwanted overhead.
(2) Prevent the other scheduling classes from changing the CPU frequency when there is some real-time load. Even if viable, this approach would be seldom accepted by the kernel community (who traditionally is more concerned with energy efficiency rather than precise real-time execution).
(3) Use the current value of the CPU frequency at the moment of the accounting, regardless of any frequency change that may have occurred since the last accounting operation. This — of course slightly inaccurate — approach is the one that has been suggested by the kernel community.

Currently, a reliable information about the actual CPU frequency is available to the schedutil governor only on ARM platforms (through a patch recently merged into the mainline kernel). This means that, at the current state, the proposed DVFS mechanism can work on non-ARM platforms only for tasks explicitly requesting CPU reclaiming (i.e., using the `SCHED_FLAG_RECLAIM` flag). Note, however, that ARM-based platforms represent the vast majority of modern mobile devices.

4.3 Kernel thread

The schedutil governor uses a worker kernel thread for driving frequency changes on platforms that do not have fast switching capabilities. In the mainline kernel, this thread is currently scheduled with the SCHED_FIFO policy and a priority equal to (MAX_USER_RT_PRIO / 2). However, this kernel thread must have higher priority than all the `SCHED_DEADLINE` tasks, otherwise a CPU-hungry task would be able of delaying the frequency switches. Hence, as a temporary workaround, the priority of this task has been raised to the maximum possible value[4].

The kernel community expressed a bit of concern due to unwanted scheduling behaviors that may happen when mixing such a high priority kernel thread with priority inheritance or similar resource sharing protocols. However, these are considered corner cases, and this temporary solution seems to be accepted waiting for a more general approach.

5 EXPERIMENTAL EVALUATION

To validate the proposed scheduler we have performed a set of extensive tests using a Freescale Sabre board based on a quad-core Cortex-A9 ARM SoC that can run at three different frequencies: 396 MHz, 792 MHz and 996 MHz. The real-time load has been generated through the rt-app framework [1], implementing one or more real-time tasks composed by periodic jobs with a fixed execution time, run through the ARM's LISA toolkit [6]. To have consistent data, all the provided values were averaged over 10 consecutive runs.

The energy consumption has been measured through the Baylibre's ACME Cape board [8] integrated with LISA. We highlight the fact that the measured values are related to the energy consumed by the whole embedded board, not just the SoC.

The first set of experiments consisted in a periodic task with period 100*ms* and execution time C, scheduled by `SCHED_DEADLINE`

[4] Technically, it has been transformed into a special SCHED_DEADLINE task without the traditional reservation values (i.e., runtime and period) and thus executed for all the needed time.

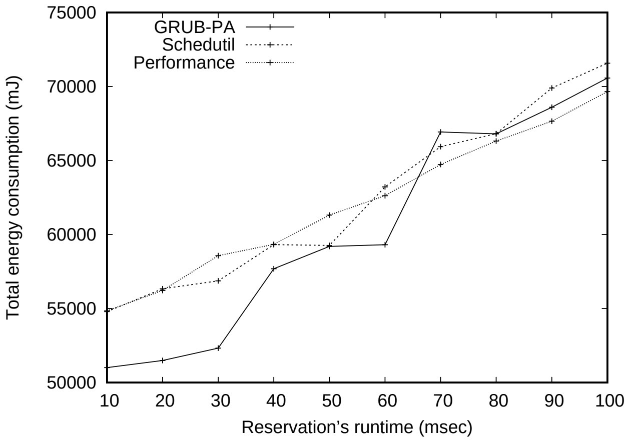

Figure 2: Energy consumption of test1 (one task, runtime 90%, period 100 msec).

with a runtime Q such that $C = 0.9Q$. Multiple experiments have been performed, with the runtime Q ranging from $10ms$ to $100ms$ (and hence C ranging from $9ms$ to $90ms$). The code base has been based on the "tip" repository [15], which contains the next version of the scheduler core. The results in terms of both energy consumption and deadline misses have been measured for the current schedutil governor, for the performance governor (which keeps the CPU always at the maximum frequency) and for the proposed modification of schedutil implementing the GRUB-PA algorithm. Figure 2 shows the average energy consumption measured by the energy meter for different values of the reservation. The registered percentage of deadline misses is summarized in Table 1. Such figures show that the performance governor has the lowest amount of deadline misses, but also the poorest energy efficiency. This behavior was of course expected. A more interesting comparison is between the current schedutil and the proposed GRUB-PA implementation. GRUB-PA, in fact, allows to considerably reduce the energy consumption when the bandwidth is lower than 70% and especially to significantly improve the real-time performance for all the values of the bandwidth. In particular, our implementation allows to have real-time performance similar to the performance governor without a huge loss in terms of energy efficiency.

A second set of experiments further stressed the deadline scheduler by setting the execution time C of the jobs exactly equal to runtime Q of the the reservation serving the task. The experimental results, shown in Figure 3 and Figure 4, respectively, confirm the behavior already illustrated. Also note that all schedulers (including the performance governor, which does not perform frequency scaling) showed a quite large amount of deadline misses. These figures therefore suggest to over-allocate the reservation with respect to the actual task needs, even when using the default governors.

A third set of experiments aimed at investigating the behavior of the scheduler when reducing the timing granularity. We have thus reduced the task period (and the SCHED_ DEADLINE reservation period to $10ms$), with the reservation's runtime Q ranging from $1ms$ to $10ms$ and the jobs execution time C (equal to $0.9Q$) ranging from $0.9ms$ to $9ms$. The results in Figure 5 show a significant gain in

Table 1: Deadline miss of test1 (one task, runtime 90%).

Resv. runtime	GRUB-PA	Performance	Schedutil
10%	0.1%	0.1%	33.5%
20%	0.0%	0.2%	44.2%
30%	0.1%	0.0%	38.1%
40%	0.1%	0.1%	32.2%
50%	0.0%	0.0%	47.7%
60%	0.0%	0.0%	22.4%
70%	0.0%	0.1%	14.7%
80%	0.0%	0.0%	12.9%
90%	0.1%	0.1%	9.4%
100%	0.4%	0.0%	16.6%

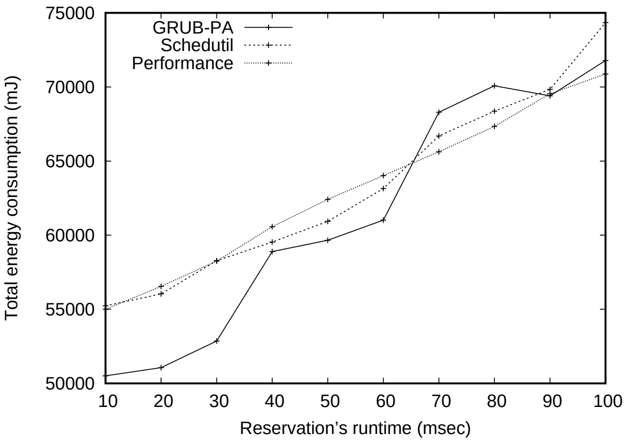

Figure 3: Energy consumption of test2 (one task, runtime 100%, period 100 msec).

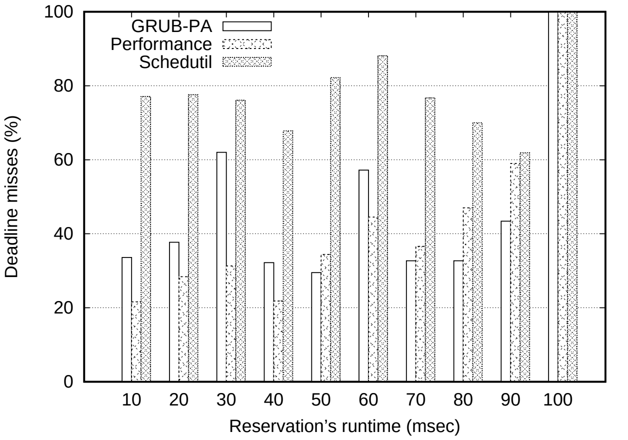

Figure 4: Deadline misses of test2 (one task, runtime 100%, period 100 msec).

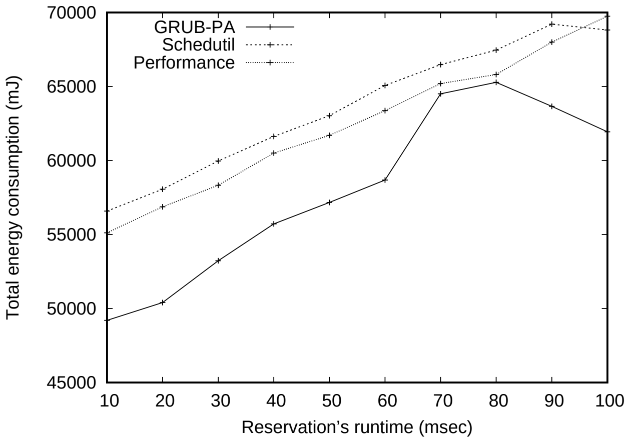

Figure 5: Energy consumption of test3 (one task, runtime 90%, period 10 msec).

terms of energy efficiency. Looking at the distribution of the misses with respect to the reservation's utilization (reported in Table 2 and plotted in Figure 6), we can see that GRUB-PA has a worse number of misses than the current schedutil governor only for values of the reservation bandwidth higher than 90%; the default schedutil governor, instead, presents a non-negligible percentage of misses across the whole range of values. Again, the average percentage of misses is lower using GRUB-PA than the governor currently available in the mainline kernel.

The difference between the theoretical GRUB-PA behavior (it should cause 0 deadline misses) and the actual behavior (noticeable deadline miss percentage for utilization $> 50\%$) is due to the fact that the physical CPU needs some time to switch the frequency (on the board used for the experiments, it is about $1ms$). This can be accounted for by increasing the SCHED_DEADLINE runtime used for scheduling the task: if the job execution time is C, the runtime has to be set to $Q = C + \epsilon$, where ϵ is the frequency switch time. Of course, this pessimistic assignment of the scheduling parameters will admit less SCHED_DEADLINE tasks in the system (for example, if $\epsilon = 3ms$ a task with job execution time $C = 8ms$ and period $P = 10ms$ will be rejected); on the other hand, it allows to respect more deadlines (making the GRUB-PA performance comparable with the ones of the performance governor for what concerns deadline misses). Some additional experiments confirmed the effectiveness of this approach.

The next sets of experiments aimed at investigating the performance when increasing the number of real-time tasks. The fourth test run four SCHED_DEADLINE tasks, equal to the number of the available cores. The four tasks, encapsulated in different reservations, had a period of $100ms$ and execution time equal to 90% of their reservation's runtime, similarly to the first test. The experimental results, shown in Figure 7 and Table 3, confirm the behavior already illustrated. Additionally, we experienced a deadlock of the target using the schedutil governor and an utilization equal to 100%.

Finally, the fifth set of experiments aimed at investigating the behavior of the scheduler with generic tasksets (composed by an

Table 2: Deadline misses of test3 (one task, runtime 90%, period 10 msec).

Resv. runtime	GRUB-PA	Performance	Schedutil
10%	0.4%	0.1%	19.2%
20%	0.1%	0.1%	17.1%
30%	0.1%	0.1%	13.8%
40%	0.4%	0.1%	13.3%
50%	0.7%	0.1%	15.8%
60%	2.1%	0.1%	8.4%
70%	2.6%	0.1%	11.4%
80%	3.6%	0.1%	14.2%
90%	52.6%	0.1%	13.6%
100%	55.4%	0.2%	33.3%

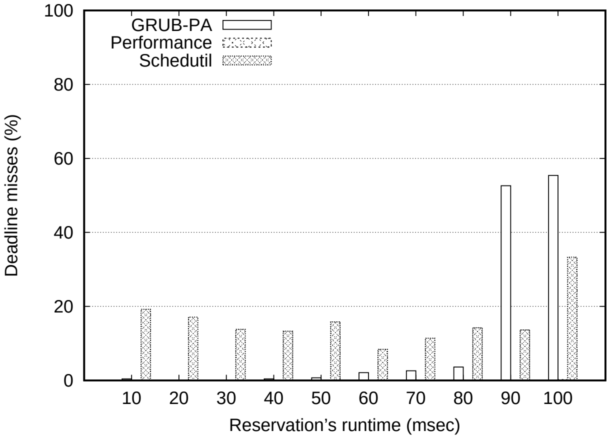

Figure 6: Deadline misses of test3 (one task, runtime 90%, period 10 msec).

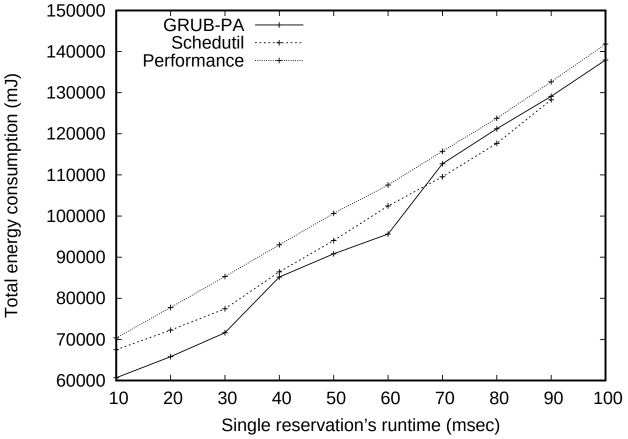

Figure 7: Energy consumption of test4 (four tasks, runtime 90%, period 100 msec).

Table 3: Deadline misses of test4 (four tasks, runtime 90%, period 100 msec).

Resv. runtime	GRUB-PA	Performance	Schedutil
10%	0.1%	0.1%	40.3%
20%	0.2%	0.1%	40.1%
30%	0.9%	0.1%	40.2%
40%	0.2%	0.1%	24.0%
50%	0.3%	0.1%	17.6%
60%	0.6%	0.2%	15.7%
70%	0.4%	0.3%	8.9%
80%	0.5%	0.6%	10.8%
90%	0.5%	0.5%	7.6%
100%	0.7%	0.5%	N/A

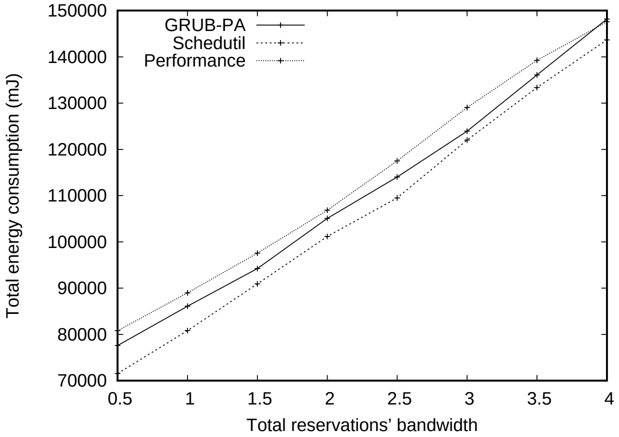

Figure 8: Energy consumption of test5 (eight tasks, runtime 90%).

even higher amount of real-time tasks). We have therefore generated sets of eight reservations with heterogeneous values of runtime and period, using the Randfixedsum algorithm [9]. Each reservation has been used to serve a real-time task with a runtime equal to 90% of its reservation's runtime. The measured values averaged over 10 consecutive runs with different sets of reservations are shown in Figure 8 and Figure 9. Several patterns can be observed in these figures:

- In terms of energy efficiency, the proposed scheduler does not perform as well as the current schedutil governor (even if still better than the performance governor).
- The real-time performance, however, was significantly improved, as the default governor had a high percentage of deadline misses even for low real-time loads.
- For very high values of the reservations' bandwidth, all the schedulers tend to show almost the same amount of deadline misses.

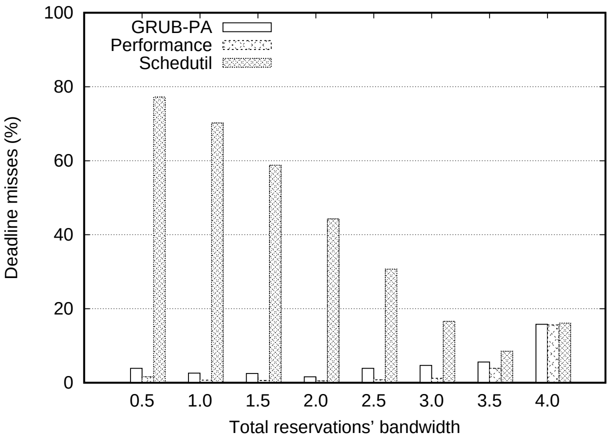

Figure 9: Deadline misses of test5 (eight tasks, runtime 90%).

6 CONCLUSIONS

In this paper we have described an implementation of the GRUB-PA energy-aware real-time scheduling algorithm in the Linux kernel, illustrating the design issues and the main choices that we have faced when implementing a theoretical scheduling algorithm in a real operating system.

The experimental results measured on a real multi-core ARM platform have shown the limits in terms of real-time performance of the schedutil governor currently available in the Linux kernel. They also confirmed that the proposed approach allows real-time performance similar to the performance governor but with a lower energy consumption.

The next step will focus on testing the scheduler on more complex platforms (e.g., the ARM big.LITTLE architecture) with a higher number of OPPs.

ACKNOWLEDGMENTS

This work has been partially funded by the European Commission through the HERCULES (H2020 GA-688860) and the RETINA (EUROSTARS E10171) projects.

REFERENCES

[1] [n. d.]. rt-app framework. https://github.com/scheduler-tools/rt-app. ([n. d.]).

[2] Luca Abeni and Giorgio Buttazzo. 1998. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. Madrid, Spain.

[3] Luca Abeni, Juri Lelli, Claudio Scordino, and Luigi Palopoli. 2014. Greedy CPU reclaiming for SCHED_DEADLINE. In *Proceedings of the 9th Real-Time Linux Workshop*. Dusseldorf, Germany.

[4] Luca Abeni, Giuseppe Lipari, Andrea Parri, and Youcheng Sun. 2016. Multicore CPU Reclaiming: Parallel or Sequential?. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*. ACM, New York, NY, USA, 1877–1884. https://doi.org/10.1145/2851613.2851743

[5] Luca Abeni, Luigi Palopoli, Claudio Scordino, and Giuseppe Lipari. 2009. Resource Reservations for General Purpose Applications. *IEEE Transactions on Industrial Informatics* 5, 1 (2009), 12–21.

[6] ARM. [n. d.]. LISA testing framework. https://github.com/ARM-software/lisa. ([n. d.]).

[7] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. 2004. Power-Aware Scheduling for Periodic Real-Time Tasks. *IEEE Trans. Comput.* 53, 5 (May 2004), 584–600.

[8] Baylibre. [n. d.]. ACME. http://baylibre.com/acme/. ([n. d.]).

[9] Paul Emberson, Roger Stafford, and Robert I Davis. 2010. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*. 6–11.

[10] G.Lipari and S.K. Baruah. 2000. Greedy reclaimation of unused bandwidth in constant bandwidth servers. In *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*. Stokholm, Sweden.

[11] Pavan Kumar and Mani Srivastava. 2000. Predictive Strategies for Low-Power RTOS Scheduling. In *Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers & Processors (ICCD '00)*. Austin, Texas, USA.

[12] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. 2016. Deadline scheduling in the Linux kernel. *Software: Practice and Experience* 46, 6 (2016), 821–839.

[13] C. L. Liu and J. Layland. 1973. Scheduling alghorithms for multiprogramming in a hard real-time environment. *J. ACM* 20, 1 (1973).

[14] J. R. Lorch and A. J. Smith. 2001. Improving Dynamic Voltage Scaling Algorithms with PACE. In *In Proceedings of the ACM SIGMETRICS 2001 Conference*. Cambridge, MA.

[15] Ingo Molnar. [n. d.]. Linux kernel tip tree. https://git.kernel.org/pub/scm/linux/kernel/git/mingo/tip.git/. ([n. d.]).

[16] Padmanabhan Pillai and Kang G. Shin. 2001. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proceeding of the 18th ACM Symposium on Operating Systems Principles*.

[17] Johan Pouwelse, Koen Langendoen, and Henk Sips. 2001. Dynamic Voltage Scaling on a Low-Power Microprocessor. In *7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*.

[18] Johan Pouwelse, Koen Langendoen, and Henk Sips. 2001. Energy Priority Scheduling for Variable Voltage Processors. In *Int. Symposium on Low Power Electronics and Design (ISLPED)*.

[19] A. Qadi, S. Goddard, and S.; Farritor. 2003. A dynamic voltage scaling algorithm for sporadic tasks. In *Proceedings of the 24th Real-Time Systems Symposium*. Cancun, Mexico, 52 – 62.

[20] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. 1998. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*.

[21] Saowanee Saewong and Ragunathan Rajkumar. 2003. Practical Voltage-Scaling for Fixed-Priority RT-Systems. In *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposiuam (RTAS)*.

[22] Claudio Scordino. 2007. *Dynamic Voltage Scaling for Energy-Constrained Real-Time Systems*. Ph.D. Dissertation. University of Pisa.

[23] Claudio Scordino and Giuseppe Lipari. 2006. A Resource Reservation Algorithm for Power-Aware Scheduling of Periodic and Aperiodic Real-Time Tasks. *IEEE Trans. Comput.* 55, 12 (2006), 1509–1522.

[24] Y.Zhu and F.Mueller. 2004. Feedback EDF Scheduling Exploiting Dynamic Voltage Scaling. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*. Toronto, Canada.