SlackCheck: A Linux Kernel Module to Verify Temporal Properties of a Task Schedule

Michele Castrovilli[1]
University of Turin, Italy

Enrico Bini
University of Turin, Italy

Abstract

The Linux Kernel offers several scheduling classes. From `SCHED_DEADLINE` down to `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`, the scheduling classes can provide different responsiveness to very diverse user workloads. Still, Linux does not offer any mechanism to take some action upon the violation of temporal constraints at runtime. The lack of such a feature is also due to the difficulty of extending the established notion of deadline to workloads which are not releasing periodic/sporadic jobs.

Exploiting the notion of supply functions for any resource schedule, we implemented SlackCheck, a kernel module which is capable to verify at runtime if a given task is assigned a desired amount of resource or not. SlackCheck adds a constant-time check at every scheduling decision and leverages the recent availability of a Runtime Verification engine in the kernel.

2012 ACM Subject Classification Software and its engineering → Scheduling; Theory of computation → Verification by model checking; Proper nouns: People, technologies and companies → Linux; Computer systems organization → Real-time operating systems; Software and its engineering → Operating systems

Keywords and phrases Linux scheduler, Runtime verification, bounded-delay resource partition, supply function, service curve, real-time calculus, network calculus

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2024.2

Funding This work has been partially supported by the spoke "FutureHPC and BigData" of the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union – NextGenerationEU.

1 Introduction

Linux is a multi-purpose operating system as it is used on desktop/laptop machines, in data-centers as hypervisor, as well as in embedded devices. Given the diversity of the target applications, its scheduler responds to the needs of different types of workloads: from compute intense to highly interactive (low latency), from constant demands to highly variable workloads triggered by the users. The developers' answer to these different requirements takes the form of different *scheduling classes*.

In the Linux kernel, the scheduling classes are arranged in a hierarchy: a task is eligible to run only if no other tasks belonging to a higher scheduling class is available. In kernel version 6.6.0 [2], the scheduling classes available to the user are (from higher to lower priority on the hierarchy):

1. `SCHED_DEADLINE` [22, 28] implementing the Constant Bandwidth Server (CBS) [2] and then scheduling the servers by EDF.

[1] corresponding author
[2] Released on 2023-10-29

2. SCHED_FIFO and SCHED_RR implementing a Fixed-Priority Scheduler, with SCHED_RR pushing back to the tail of the ready queue any task running longer than a given *round* (whose default length is 100 milliseconds[3]).
3. SCHED_OTHER and SCHED_BATCH, implementing the Completely Fair Scheduler (CFS) [42].

As demonstrated by the merge into mainline of SCHED_DEADLINE, the kernel maintainers have shown an increasing attention to the real-time constraints of workloads [48]. Still, no run-time mechanism exists to monitor the amount of resource supplied to a given task. In fact, when tasks are scheduled by SCHED_DEADLINE, the deadline is indeed used to schedule tasks, but no (recovery or else) action is possible upon a deadline miss.

In this work, we have developed SlackCheck, a kernel module which monitors and verifies if the time allocated to any tasks conforms to a given *supply function* (more details on this model are reported in Section 2). Upon user's choice, when such constraint is violated this mechanism can either warn the user with a kernel message, try to recover from the situation, or stop the system entirely. Other possible reactions can be implemented, such as retuning the process within the scheduler either by changing its priority or scheduling class. This allows for a great flexibility thanks to the underlying Runtime Verification system.

1.1 Related works

The attempt to execute real-time applications over Linux is not new. Many works have addressed the latency added by the kernel itself. Abeni et al. [4] analyzed the timer resolution of an x86 platform and defined a cyclictest-like metric to evaluate the latency due to non-preemptible sections in the OS. Cerqueira and Brandenburg [15] compared the scheduling latencies of PREEMPT_RT with LITMUS^{RT}, which is a real-time, Linux-based testbed, developed for empirically evaluating multiprocessor real-time scheduling algorithms [14]. Cinque et al. [17] proposed real-time containers, implemented over RTAI[4]. To respond to the needs of applications with strict timing, the implementation of table-driven scheduling in Linux was proposed [25]. Recently, rtla osnoise was presented, a per-CPU tracer that enables the tracing of the sources of the noise, facilitating the analysis and debugging of the system [11]. A formal verification approach for the Linux kernel, based on automata models, was proposed by Bristot et al. [12].

Monitoring the progress of applications and possibly take corrective actions was proposed by several authors. A first proposal for feedback-based scheduling of time sharing systems was proposed by Corbató et al. [18]. Stankovic and his former students proposed guaranteed composition of schedulers [38] as well as model and analysis for feedback scheduling [31]. A similar approach was proposed by Cervin et al. [16]. Abeni et al. [3] proposed adaptive reservations to adjust to the varying loads. All the mentioned works, however, did need a mechanism to explicitly signal the deviation from the correct behavior (such as the occurrence of deadline misses). Our work, instead, does not require any explicit mechanism as it is based on the always-available scheduling events.

Supply functions have been used, possibly under different names in different contexts (e.g. *service curves* in network/real-time calculus [19, 8, 43, 27]), to model the availability of different types of resource: processing time of a single [34, 30, 40] or multi-core platform [39, 10, 29, 46, 13], network [19, 7], memory [47, 1], and I/O [37]. In words, a supply function represents a lower bound to the amount of resources allocated by a scheduler (in Section 2, we recall the necessary formalism).

[3] https://elixir.bootlin.com/linux/v6.6.9/source/include/linux/sched/rt.h#L57. Accessed on January 7th, 2024.
[4] https://www.rtai.org/. Accessed on February 10th, 2024.

A simple form of a supply function is expressed by a linear lower bound and modeled by the slope and the intercept of the line with the x-axis, often called *bandwidth* and *delay* in real-time systems. Without aiming at an exhaustive coverage of this vast research area, the following works have had a significant impact:

- Cruz [19] introduced the "(σ, ρ) regulator" to control bandwidth and delay of a given flow of packets,
- Parekh and Gallager introduced the concept of Generalized Processor Sharing (GPS) as a means to fairly allocate resources among competing flows [36],
- In network calculus, rate-latency service curves model precisely the same type of supply [41, 27],
- Mok et al. [34] introduced the notion of "least supply function" and "bounded-delay resource partition" (α, Δ), which we will be using in this paper,
- Mostly inspired by [34], several authors [30, 40, 6] derived schedulability conditions for a set of tasks to be scheduled over a partially available CPU modeled by a supply function.

The model of resource schedules by supply functions (or other equivalent models possibly named differently) did influence the development of the Linux kernel and other OSes. Stoica et al. [42] developed a real-time scheduling policy over FreeBSD 2.0.5. Rialto [24] was an OS developed at Microsoft Research offering real-time guarantees by a pre-computed schedule. Oikawa and Rajkumar implemented a resource kernel to ensure predictable timing across diverse hardware and operating systems [35]. Wang and Lin [45] developed some real-time (hierarchical) scheduler on top of Linux, kernel version 2.0.35. Regehr and Stankovic [38] implemented a hierarchical scheduler with soft real-time guarantees over Windows 2000 OS.

In the virtualization context, many advancements were made. Xtratum is an hypervisor for embedded systems [33]. Cucinotta et al. provided real-time guarantees for both the processing and the networking [20]. RT-Xen [46] was implementing Xen virtual CPUs by time partitions. Maggio et al. [32] developed a tool to measure supply functions of any given execution platform, possibly with some degree of parallelism. However, their work was based on some off-line analysis of scheduling traces. Hermes was proposed as hypervisor for microcontrollers without MMU [26].

In summary, the supply function model has driven the development of many scheduler modifications over the years. Pushing these ideas upstream to the mainline kernel, however, is not simple as the kernel scheduler needs to account for many real-world details, different use cases and scenarios. This motivates our approach of proposing a stand alone kernel module which monitors and detects violations of temporal constraints, independently of the scheduler choice.

Contribution of this paper

First, we have determined a necessary and sufficient condition for the verification of a linear lower bound to the service received by the task. Then, we have implemented a Linux module to monitor and verify temporal constraints expressed by linear supply lower bound functions. Our method has $O(1)$ time complexity, as highly desirable (i.e. practically mandatory) for kernel code invoked at scheduling decisions.

2 Model of schedule and constraints

SlackCheck can track the execution of a task and check if any *temporal constraint* is violated at runtime. This section illustrates the formalism used to represent schedules and temporal constraints.

SlackCheck is attached to a task, which we denote by τ. Also, let us denote by:

- $[\mathsf{in}_k]_{k\in\mathbb{N}} = [\mathsf{in}_0, \mathsf{in}_1, \mathsf{in}_2, \ldots]$ the sequence of instants when τ starts to be scheduled, called *sched-in* instants for brevity, and
- $[\mathsf{out}_k]_{k\in\mathbb{N}}$ the sequence of instants when τ is removed from a CPU, called *sched-out* instants.

We assume that

- $\mathsf{out}_0 = 0$, meaning that we count the time from the first sched-out
- $\mathsf{in}_0 \geq \mathsf{out}_0$, meaning the first sched-in follows the first sched-out. If τ is scheduled at time 0, then we have $\mathsf{in}_0 = \mathsf{out}_0$.
- $\forall k \in \mathbb{N}$ we have $\mathsf{in}_k < \mathsf{out}_{k+1} < \mathsf{in}_{k+1}$, because sched-in and sched-out instants obviously alternate over the task τ lifetime.

Also, to exclude Zeno's sequences, in which both sched-in and sched-out instants have an accumulation point, we require that $\lim_k \mathsf{in}_k = \infty$. This implies that the same property holds for the sched-out instants as well.

Given the instants in_k and out_k as defined above, we define the *schedule* function by

$$\forall k \in \mathbb{N}, \qquad s(t) = \begin{cases} 0 & \text{when } \mathsf{out}_k \leq t < \mathsf{in}_k \\ 1 & \text{when } \mathsf{in}_k \leq t < \mathsf{out}_{k+1}. \end{cases} \tag{1}$$

Also, the cumulative amount of service received by τ over any interval $[a, b]$ is denoted by

$$\mathsf{sched}(a, b) = \int_a^b s(t)\, dt. \tag{2}$$

The *supply (lower) bound function* $\mathsf{sbf}(t)$ for the schedule $s(t)$ of the task τ is a function such that [34, 30, 40]

$$\forall t_0, t \geq 0, \quad \mathsf{sbf}(t) \leq \mathsf{sched}(t_0, t_0 + t), \tag{3}$$

meaning that $\mathsf{sbf}(t)$ is a lower bound to the amount of resource allocated by the scheduler to τ in any interval of length t. Many different functions $\mathsf{sbf}(t)$ can fulfill (3) as, for example, the constant $\mathsf{sbf}(t) = 0$. It is, however, of greater practical interest to have the $\mathsf{sbf}(t)$ be the largest possible satisfying (3). For example, if $s(t)$ represents any schedule of a periodic server allocating a budget Q with period P, then a valid $\mathsf{sbf}(t)$ is

$$\mathsf{sbf}(t) = \max\{0,\ t - P + Q - (k + 1)(P - Q),\ k\,Q\}, \qquad \text{with } k = \left\lceil \frac{t - P + Q}{P} \right\rceil \tag{4}$$

which is represented in Figure 1. We underline that the notion of supply function is analogous to service curves of network/real-time calculus [19, 43, 27], as mentioned earlier in Section 1.1.

Another typical form of the $\mathsf{sbf}(t)$ to bound from below the amount of allocated resource is the so-called *bounded-delay* partition [34], which requires the definition of two parameters

- the *bandwidth* α, and
- the *delay* Δ.

These parameters enable the definition of the *supply linear bound function*

$$\mathsf{slbf}(t) = \max\{0,\ \alpha(t - \Delta)\}, \tag{5}$$

and we say that a resource schedule $s(t)$ complies with the bandwidth-delay pair (α, Δ) whenever

$$\forall t_0, t_1 \geq t_0, \quad \mathsf{sched}(t_0, t_1) \geq \mathsf{slbf}(t_1 - t_0). \tag{6}$$

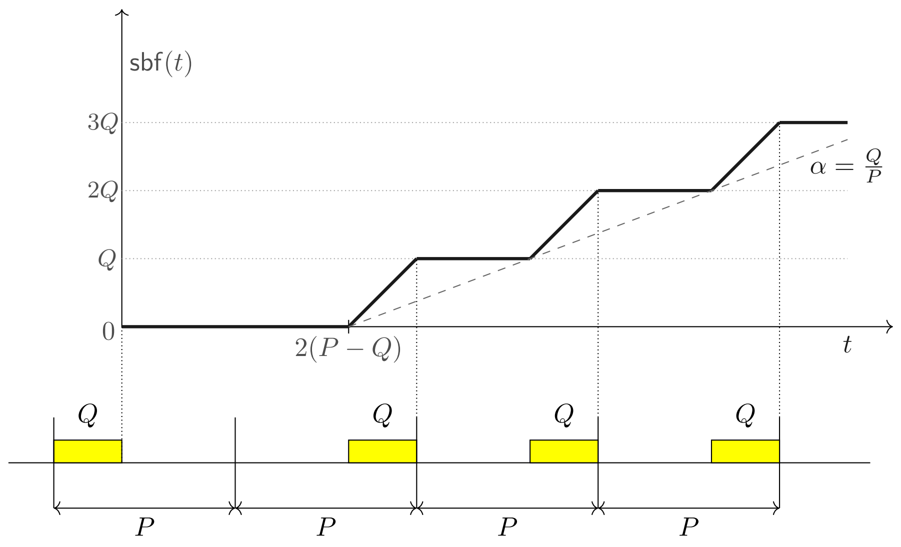


Figure 1 Example of supply bound function for a periodic server.

For example, any periodic server with the supply of (4) has a supply linear lower bound with a bandwidth $\alpha = \frac{Q}{P}$ and a delay $\Delta = 2(P - Q)$. We choose this form of temporal constraints because it can be applied to any schedule $s(t)$, regardless of the type of workload generating such a schedule. Indeed other forms of constraints exist such as the deadline or the number of tolerable deadline misses among consecutive job releases. These constraints, however, require the task to be periodic, whereas SlackCheck can be applied to any workload and any scheduling policy.

Hence, from now on, whenever we say "a workload meets the temporal constraints", we mean that the schedule of the task τ satisfies Equation (6), which is the form of our temporal constraint.

3 Verification of the linear supply lower bound in $O(1)$

The goal driving our work is the implementation of a mechanism which is capable to detect at runtime if the schedule of a given task τ violates Equation (6). As the equation shows, the condition needs to be enforced for any start and end times t_0, t_1 of the interval $[t_0, t_1]$. A straightforward implementation of the condition of (6) is clearly infeasible because:

- it needs to be constantly monitored at any time t_1, and
- for any t_1, it needs to be checked for any t_0 in the past ($t_0 \leq t_1$).

The two above reasons are difficult to implement, as constantly checking for each time t_1 would add too much overhead in the kernel. For the latter reason, the number of points t_0 would grow with the passing of time, therefore the straightforward check would also take too space in memory to be implementable. Of course, these issues apply to all tasks, in case SlackCheck is implemented for many tasks.

In this section, we describe an algorithm which is executed only at a discrete set of instants and has constant $O(1)$ time and space complexity. We remark that this characteristic is essential for code which needs to be executed at the frequency of the scheduling events in the kernel.

In short, our algorithm:

- checks the condition of (6) only at sched-in instants t_1, and
- updates in constant time, an internal state variable of *slack*, which keeps track of the most constraining t_0 in the past.

The slack is the time remaining until the latest instant for the task τ to be scheduled in order not to violate the condition of (6).

Section 3.1 illustrates the details of the algorithm for the slack calculation, whereas Section 3.2 provides the proof of correctness of such an algorithm.

3.1 Algorithm

As mentioned earlier, our algorithm revolves around the notion of *slack*, which is formally defined next.

▶ **Definition 1.** *Let* $[\mathsf{in}_k]_{k\in\mathbb{N}}$ *and* $[\mathsf{out}_k]_{k\in\mathbb{N}}$ *be the sequences of sched-in and sched-out instants, respectively.*

We define slack *of the schedule of the task* τ*, as follows*

$$\mathsf{slack}(\mathsf{out}_0) = \Delta \tag{7}$$

$$\mathsf{slack}(\mathsf{in}_k) = \mathsf{slack}(\mathsf{out}_k) - (\mathsf{in}_k - \mathsf{out}_k) \tag{8}$$

$$\mathsf{slack}(\mathsf{out}_k) = \min\{\Delta, \mathsf{slack}(\mathsf{in}_{k-1}) + \frac{1-\alpha}{\alpha}(\mathsf{out}_k - \mathsf{in}_{k-1})\}. \tag{9}$$

The physical interpretation of the $\mathsf{slack}(t)$ is the amount of time the task τ can tolerate without being scheduled before violating the constraint of the linear lower bound of Equation (6). In fact, the next theorem establishes a useful equivalence.

▶ **Theorem 2.** *Let* $\mathsf{slack}(t)$ *be the slack as defined in Def. 1. Then, the condition of (6) is equivalent to*

$$\forall h \in \mathbb{N}, \quad \mathsf{slack}(\mathsf{in}_h) \geq 0. \tag{10}$$

Before proving the theorem (the proof of Theorem 2 is in Section 3.2), we illustrate its advantages and application.

From the computational perspective, the combination of Theorem 2 and Definition 1 can be implemented by:

- Updating a state variable which tracks the value of $\mathsf{slack}(t)$. This should happen at both sched-in and sched-out instants.
- It is also necessary to check the condition of Eq. (10) at sched-in instants.

Both statements can be made in constant time complexity and constant space complexity, as they required to store the only variable of the slack.

Let us show an example. In Figure 2, we show the schedule of a task (in yellow) as well as the value of $\mathsf{slack}(t)$. We arbitrarily choose $\alpha = \frac{2}{3}$ and $\Delta = 4$. At the first sched-out $\mathsf{out}_0 = 0$, we initialize $\mathsf{slack}(\mathsf{out}_0) = \Delta$ (equal to 4 in the example) as required by Eq. (7). Then, during idle intervals, $\mathsf{slack}(t)$ decreases at the rate of one, as it follows from (8). When the task τ is running, $\mathsf{slack}(t)$ increases at the rate of $\frac{1-\alpha}{\alpha}$ as indicate by Eq. (9), which is $\frac{1}{2}$ in the example. Also, we remark that $\mathsf{slack}(t)$ cannot grow more than Δ and it is always saturated by Δ, otherwise by running for long enough time, we may allow arbitrary long idle intervals. Such a saturation is represented by the dotted line which grows beyond Δ, but does not affect the value of $\mathsf{slack}(\mathsf{out}_1)$. As the schedule progresses, a negative slack is detected at time $t = \mathsf{in}_3 = 15$, meaning that the constraint on the supply lower bound function of (6) is violated.

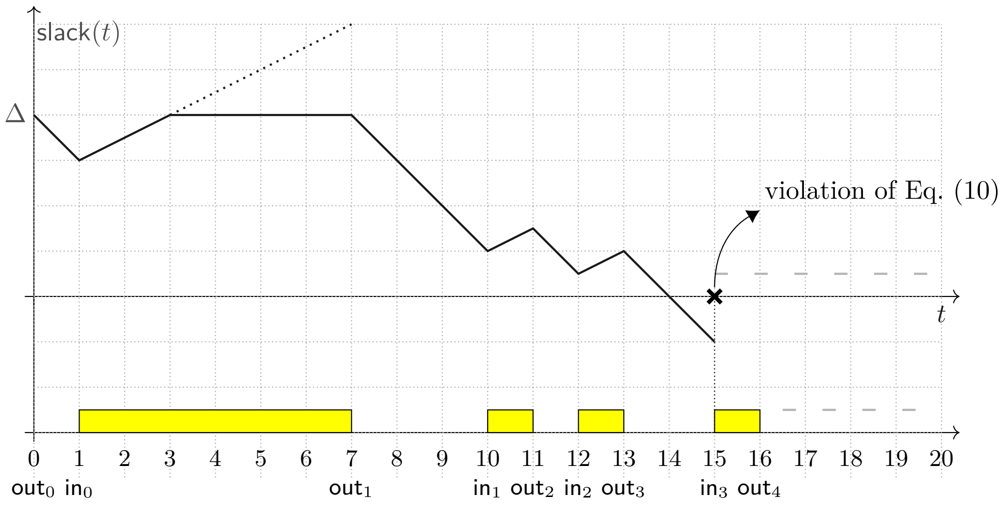


Figure 2 An example of detection of a constraint violation. In the example, we choose a supply linear bound slbf(t) given by $\alpha = \frac{2}{3}$ and $\Delta = 4$. The violation of Equation (10) is detected at the sched-in instant $\mathsf{in}_k = 15$. In fact, the condition of Eq. (6) is also violated for $t_0 = 7$ and $t_1 = 15$ as we have $\mathsf{sched}(7, 15) = 2 < \mathsf{slbf}(15 - 7) = \alpha(8 - \Delta) = \frac{2}{3} \times 4 = 2 + \frac{2}{3}$.

3.2 Proof of correctness

In this section we prove Theorem 2 and discuss its implications.

Proof of Theorem 2. The goal is to demonstrate that Eq. (6) is equivalent to Eq. (10).

First, we determine that (6) is equivalent to

$$\forall k, h \in \mathbb{N}, h \geq k, \quad \mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) \geq \mathsf{slbf}(\mathsf{in}_h - \mathsf{out}_k). \tag{11}$$

The implication (6) ⇒ (11) is obvious because if (6) is true for all real numbers, it is also true for the subset of sched-in/sched-out instants, as needed by (11).

The implication (11) ⇒ (6) requires slightly more efforts. We are given any t_0 and t_1, with $t_1 \geq t_0$ and we are building out_k and in_h to exploit the property of (11). From t_0, we build out_k as follows

$$\mathsf{out}_k = \begin{cases} \max\{\mathsf{out}_\ell : \mathsf{out}_\ell \leq t_0\} & \text{if } s(t_0) = 0 \\ \min\{\mathsf{out}_\ell : \mathsf{out}_\ell \geq t_0\} & \text{if } s(t_0) = 1, \end{cases}$$

with $s(t)$, schedule function as defined in (1). In words, out_k is the latest sched-out instant preceding t_0 if τ is scheduled at t_0, or the earliest sched-out after t_0 otherwise. Also, we build in_h from t_1 as follows:

$$\mathsf{in}_h = \begin{cases} \min\{\mathsf{in}_\ell : \mathsf{in}_\ell \geq t_1\} & \text{if } s(t_1) = 0 \\ \max\{\mathsf{in}_\ell : \mathsf{in}_\ell \leq t_1\} & \text{if } s(t_1) = 1. \end{cases}$$

The minima and maxima above always exist because they are taken over a non empty set with no accumulation point.

We consider all four cases for $s(t_0)$ and $s(t_1)$. If the task τ is not running at t_0 nor at t_1, that is $s(t_0) = s(t_1) = 0$, we have

$$\begin{aligned}
&\mathsf{sched}(t_0, t_1) = \mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) && \tau \text{ is idle in } [\mathsf{out}_k, t_0) \cup [t_1, \mathsf{in}_h)\\
&\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) \geq \mathsf{slbf}(\mathsf{in}_h - \mathsf{out}_k) && \text{for the hypothesis of (11)}\\
&\mathsf{slbf}(\mathsf{in}_h - \mathsf{out}_k) \geq \mathsf{slbf}(t_1 - t_0) && [t_0, t_1) \subseteq [\mathsf{out}_k, \mathsf{in}_h) \text{ and } \mathsf{slbf}(t) \text{ is non-decreasing}
\end{aligned}$$

meaning that $\mathsf{sched}(t_0, t_1) \geq \mathsf{slbf}(t_1 - t_0)$, as required by (6). If $s(t_0) = s(t_1) = 1$, then

$$\begin{aligned}
&\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) \geq \mathsf{slbf}(\mathsf{in}_h - \mathsf{out}_k) && \text{from (11)}\\
&(\mathsf{out}_k - t_0) + \mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) \geq \mathsf{slbf}(\mathsf{in}_h - \mathsf{out}_k) + \alpha(\mathsf{out}_k - t_0) && \text{because } \alpha \leq 1\\
&\mathsf{sched}(t_0, \mathsf{in}_h) \geq \mathsf{slbf}(\mathsf{in}_h - \mathsf{out}_k) + \alpha(\mathsf{out}_k - t_0) && \tau \text{ runs in } [t_0, \mathsf{out}_k)\\
&\mathsf{sched}(t_0, \mathsf{in}_h) \geq \mathsf{slbf}(\mathsf{in}_h - t_0) && \text{Def. of } \mathsf{slbf}(t) \text{ of (5)}\\
&(t_1 - \mathsf{in}_h) + \mathsf{sched}(t_0, \mathsf{in}_h) \geq \mathsf{slbf}(\mathsf{in}_h - t_0) + \alpha(t_1 - \mathsf{in}_h) && \text{because } \alpha \leq 1\\
&\mathsf{sched}(t_0, t_1) \geq \mathsf{slbf}(t_1 - t_0) && \text{same steps as above}
\end{aligned}$$

as required by (6). The other two cases follow the same steps as above. We have then concluded the equivalence between the conditions of the Equations (6) and (11).

It is now time to prove the equivalence between the statement of Theorem 2, which is Equation (10) and our condition of (11), just proved to be equivalent to (6).

We start proving that (10) implies (11), which we do by contradiction. Let k and h be two indices such that

$$\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) < \mathsf{slbf}(\mathsf{in}_h - \mathsf{out}_k) = \max\{0, \alpha(\mathsf{in}_h - \mathsf{out}_k - \Delta)\}$$

from the definition of $\mathsf{slbf}(t)$ of Eq. (5). Since $\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) \geq 0$, it must necessarily be

$$\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) < \alpha(\mathsf{in}_h - \mathsf{out}_k - \Delta). \tag{12}$$

Let us establish a relation between the slack at out_h and the slack at preceding sched-out instants.

$$\begin{aligned}
\mathsf{slack}(\mathsf{out}_h) &\leq \mathsf{slack}(\mathsf{in}_{h-1}) + \frac{1-\alpha}{\alpha}(\mathsf{out}_h - \mathsf{in}_{h-1}) && \text{from (9)}\\
&= \mathsf{slack}(\mathsf{out}_{h-1}) - (\mathsf{in}_{h-1} - \mathsf{out}_{h-1}) + \left(\frac{1}{\alpha} - 1\right)(\mathsf{out}_h - \mathsf{in}_{h-1}) && \text{from (8)}\\
&= \mathsf{slack}(\mathsf{out}_{h-1}) - (\mathsf{out}_h - \mathsf{out}_{h-1}) + \frac{1}{\alpha}(\mathsf{out}_h - \mathsf{in}_{h-1})\\
&\leq \mathsf{slack}(\mathsf{out}_k) - \sum_{\ell=k+1}^{h}(\mathsf{out}_\ell - \mathsf{out}_{\ell-1}) + \frac{1}{\alpha}\sum_{\ell=k+1}^{h}(\mathsf{out}_\ell - \mathsf{in}_{\ell-1}) && \text{recursively}\\
&\leq \Delta - (\mathsf{out}_h - \mathsf{out}_k) + \frac{1}{\alpha}\sum_{\ell=k+1}^{h}(\mathsf{out}_\ell - \mathsf{in}_{\ell-1}) && (13)
\end{aligned}$$

with the upper bound of Δ to $\mathsf{slack}(\mathsf{out}_k)$ holding from its definition of Eq. (9).

We are now ready to close this branch of the proof as we have

$$\begin{aligned}
\mathsf{slack}(\mathsf{in}_h) &= \mathsf{slack}(\mathsf{out}_h) - (\mathsf{in}_h - \mathsf{out}_h) && \text{from (8)}\\
&\le \Delta - (\mathsf{in}_h - \mathsf{out}_k) + \frac{1}{\alpha}\sum_{\ell=k+1}^{h}(\mathsf{out}_\ell - \mathsf{in}_{\ell-1}) && \text{using (13)}\\
&\le \Delta - (\mathsf{in}_h - \mathsf{out}_k) + \frac{1}{\alpha}\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h)\\
&< \Delta - (\mathsf{in}_h - \mathsf{out}_k) + \frac{1}{\alpha}\alpha(\mathsf{in}_h - \mathsf{out}_k - \Delta) = 0 && \text{hypothesis of (12)}
\end{aligned}$$

demonstrating that the slack at in_h is negative.

The last part of the proof is demonstrating that (11) implies (10), which we prove again by contradiction. Let h be an index violating (10), that is

$$\mathsf{slack}(\mathsf{in}_h) < 0. \tag{14}$$

We construct out_k as

$$\mathsf{out}_k = \max\{\mathsf{out}_\ell : \mathsf{out}_\ell \le \mathsf{in}_h \ \wedge\ \mathsf{slack}(\mathsf{out}_\ell) = \Delta\}. \tag{15}$$

In words, out_k is the latest sched-out equal to Δ and preceding in_h. Such a value always exist because the set is not empty (it contains at least out_0 which precedes any sched-in).

Let us now relate the slack for consecutive sched-out instants. From (9) and (8), for all $\ell = k+1, \ldots, h$ we have

$$\begin{aligned}
&\mathsf{slack}(\mathsf{out}_\ell) = \mathsf{slack}(\mathsf{in}_{\ell-1}) + \left(\frac{1}{\alpha} - 1\right)(\mathsf{out}_\ell - \mathsf{in}_{\ell-1}) =\\
&\mathsf{slack}(\mathsf{out}_{\ell-1}) - (\mathsf{in}_{\ell-1} - \mathsf{out}_{\ell-1}) + \left(\frac{1}{\alpha} - 1\right)(\mathsf{out}_\ell - \mathsf{in}_{\ell-1}) =\\
&\qquad\qquad \mathsf{slack}(\mathsf{out}_{\ell-1}) - (\mathsf{out}_\ell - \mathsf{out}_{\ell-1}) + \frac{1}{\alpha}(\mathsf{out}_\ell - \mathsf{in}_{\ell-1}),
\end{aligned}$$

because the definition out_k of (15) implies that for all these indices ℓ, $\mathsf{slack}(\mathsf{out}_\ell)$ is given by the second expression in the minimum of (9). By applying the relation above recursively for all $\ell = k+1, \ldots, h$ we find:

$$\mathsf{slack}(\mathsf{out}_h) = \mathsf{slack}(\mathsf{out}_k) - (\mathsf{out}_h - \mathsf{out}_k) + \frac{1}{\alpha}\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h),$$

and from the slack of a sched-in instant of (8)

$$\mathsf{slack}(\mathsf{in}_h) = \mathsf{slack}(\mathsf{out}_k) - (\mathsf{in}_h - \mathsf{out}_k) + \frac{1}{\alpha}\mathsf{sched}(\mathsf{out}_k,\mathsf{in}_h) = \Delta - (\mathsf{in}_h - \mathsf{out}_k) + \frac{1}{\alpha}\mathsf{sched}(\mathsf{out}_k,\mathsf{in}_h),$$

which also exploits that (15) implies that $\mathsf{slack}(\mathsf{out}_k) = \Delta$. Finally, from (14)

$$\begin{aligned}
&\Delta - (\mathsf{in}_h - \mathsf{out}_k) + \frac{1}{\alpha}\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) < 0\\
&\mathsf{sched}(\mathsf{out}_k, \mathsf{in}_h) < \alpha(\mathsf{in}_h - \mathsf{out}_k - \Delta) \le \mathsf{slbf}(\mathsf{in}_h - \mathsf{out}_k)
\end{aligned}$$

as required in this last case of the proof. We conclude then that (10) is equivalent to (11) which is equivalent to (6), as desired. ◀

4 Our kernel module SlackCheck

This section describes the developed kernel module, which implements the constant-time runtime verification algorithm of Section 3. Before delving into the implementation details, Section 4.1 gives a minimal background on the Linux scheduler necessary to understand the implementation and the experiments. Section 4.2 illustrates the Runtime Verification engine in Linux. Finally, Section 4.3 describes our implementation and discusses the limits due to, for example, the limits of the representation of integers.

4.1 Scheduling classes in Linux

In the Linux scheduler, tasks are partitioned among *scheduling classes*, which are sorted in a priority order: a task is eligible to run only if no other tasks belonging to a higher scheduling class is available. In priority order, these classes are:

1. `SCHED_DEADLINE`
2. `SCHED_FIFO` and `SCHED_RR`
3. `SCHED_OTHER` and `SCHED_BATCH`
4. `SCHED_IDLE`

Only `SCHED_OTHER`, `SCHED_BATCH` and `SCHED_IDLE` are available to non privileged users. `SCHED_DEADLINE`, `SCHED_FIFO` and `SCHED_RR` are reserved for the system's and root processes. It is worth mentioning a recent proposal of a time-triggered `SCHED_TT` class [25] implemented on top of the hierarchy.

`SCHED_DEADLINE` [22, 28] is at the top of the hierarchy among the scheduling classes available for processes and implements the Constant Bandwidth Server (CBS) [2], which schedules the servers by EDF with three parameters: the period `sched_period`, the periodic budget `sched_runtime`, and the deadline `sched_deadline`. The budget specifies for how long the process can run within a period, while the deadline determines the absolute deadline used by EDF to sort the runqueue. A basic utilization-based admission control is made upon the invocation of `sched_setattr` setting the `SCHED_DEADLINE` scheduling class. Such a test is implemented by making `sched_setattr` fail (with error `EINVAL`) if the total utilization of `SCHED_DEADLINE` tasks would exceed 95% of the processing capacity[5]. Such a test does not guarantee the budget to be allocated by the deadline [23], as it may be scheduled well beyond the deadline [21, 44, 5]. We remark, however, that checking exact feasibility of the `SCHED_DEADLINE` tasks, is not among the goals of kernel developers.

Below `SCHED_DEADLINE` we have two classes at the same level in the hierarchy: `SCHED_FIFO` and `SCHED_RR`. Processes queued in these classes have a priority value from 0–99. An higher priority process will preempt a lower priority one. `SCHED_FIFO` implements a Fixed-Priority Scheduler: only the highest priority one will always run, until it terminates. `SCHED_RR` instead, runs tasks with the highest scheduling priority, for a maximum of a *round* (`sched_rr_timeslice_ms`, with a default length of 100 milliseconds[6]). After the round is finished, the task is pushed back to the tail of the queue of same priority tasks. If there is a single task per priority, `SCHED_RR` is the same as `SCHED_FIFO`.

The first scheduling classes available to regular non-privileged users are `SCHED_OTHER` and `SCHED_BATCH`. They both implement the Completely Fair Scheduler (CFS). These two classes are identical and follow the same priority, with the exception that `SCHED_BATCH` processes are

[5] `man 7 sched`.
[6] `https://elixir.bootlin.com/linux/v6.6.9/source/include/linux/sched/rt.h#L57` Accessed on January 7th, 2024.

assumed to be CPU-intensive, with a small scheduling penalty incurred from not updating the process statistics as it leaves the runqueue, therefore having its priority slightly lowered, as the algorithm thinks the process is using the entirety of its timeslice instead. As of kernel 6.6, the current CFS algorithm implemented has changed from Completely Fair Scheduler (CFS) to Earliest Eligible Virtual Deadline First (EEVDF) [42].

At the bottom of the hierarchy, the `SCHED_IDLE` class runs only when when the processor is idle and no other process needs to run.

The scheduling decisions can be traced by commands such as `trace-cmd`[7], which tracks *kernel events*. A kernel event is the recording of a function call within the kernel's code. This function is recorded alongside the invocation parameters, and can be both intercepted and written to a log. When a process starts for the first time, a `sched_process_exec` kernel event is generated. This event represents the process structure being loaded into the CPU runqueue, waiting for the kernel to give it resources and processor time.

When a context switch happens, a `sched_switch` event is generated. This event has two parameters, `prev` and `next`, which point to the `task_struct` of the leaving process and the one entering. If no process was on the CPU or the CPU does not have any more tasks to run, `prev` or `next` will be respectively pointing to the idle process.

When a task ends, whether voluntarily or terminated by a signal, a `sched_process_exit` kernel event is generated, notifying that the task has finished. Both `sched_process_exec` and `sched_process_exit` carry as a parameter the pointer of the `task_struct` of the task.

The event `sched_switch` is particularly relevant for our purpose because we will be using it to determine precisely when the task under analysis is assigned a CPU and when it is instead removed from a CPU. Linking our model of Section 2 and the scheduler terminology:

- a sched-in instant in_k is the timestamp of a `sched_switch` event with the parameter `next` equal to the PID of the task τ under control, and
- a sched-out instant out_k is the timestamp of a `sched_switch` event with the parameter `prev` equal to such a PID.

4.2 Runtime verification

Our kernel module SlackCheck relies on a kernel subsystem implementing Runtime Verification. This is a lightweight technique for verifying at runtime whether a program follows a certain specification. It is based on event traces generated by the program to be verified, so that the order in which the process events happen satisfy certain temporal logic formulas [9]. In the Linux kernel, this is implemented with the generation of a deterministic automata that follows kernel events via tracing mechanisms. The state of the automata is updated in parallel with the kernel events regarding the monitored task, by interrupting the event with a constant operation, and resuming the normal handling of it after updating the model and slack value.

The runtime verification subsystem uses trace probes[8], in order to call a function whenever a probed event happens. For example, we register our probe on the `sched_switch` event to determine whether a sched-in or sched-out occurred. Then, we invoke the functions in Listings 1 and 2, accordingly.

The Linux runtime verification also allows for a system of reactors. The kernel can dynamically decide what action to take when one of the properties expressed by these

[7] `https://www.trace-cmd.org/`
[8] `https://docs.kernel.org/trace/tracepoints.html`

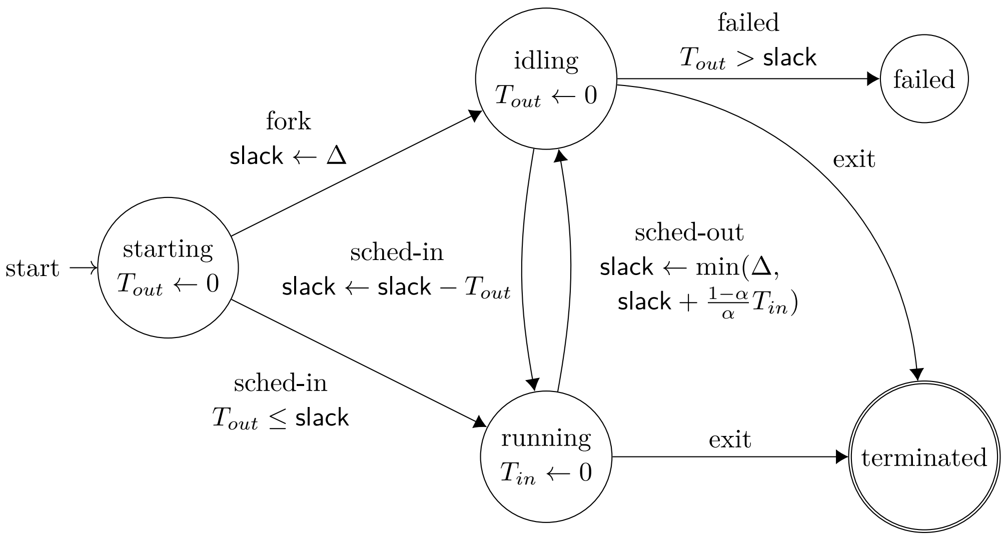

start
starting
$T_{out} \leftarrow 0$
fork
$\mathsf{slack} \leftarrow \Delta$
idling
$T_{out} \leftarrow 0$
failed
$T_{out} > \mathsf{slack}$
failed
exit
sched-in
$\mathsf{slack} \leftarrow \mathsf{slack} - T_{out}$
sched-out
$\mathsf{slack} \leftarrow \min(\Delta, \mathsf{slack} + \frac{1-\alpha}{\alpha} T_{in})$
sched-in
$T_{out} \leq \mathsf{slack}$
running
$T_{in} \leftarrow 0$
exit
terminated

Figure 3 Runtime Verification timed automata implementing SLACKCHECK. When a task is in "idling" state, it is not running, either preempted or blocked. T_{in} and T_{out} are timers that start as the automata reaches the running or idle state respectively. T_{in} is used to count the time the task τ is being scheduled, T_{out} is used to count the time the task τ is not being scheduled. As an example, when in state "idling", the timer T_{out} is reset to 0 and counts as time passes. Linking to the notation of in_k and out_k for sched-in and sched-out instants, respectively, when the k-th sched-in happens (from state "starting" or "idling") at in_k an idle interval has just finished and the timer T_{out} takes the value of $T_{out} = \mathsf{in}_k - \mathsf{out}_k$. Instead, when the k-th sched-out happens at out_k (from the "running" state) T_{in} takes the value of the length of the last running interval $\mathsf{out}_k - \mathsf{in}_{k-1}$.

automata is violated. Possible reactors vary among a simple print to the kernel buffer or causing a kernel panic, halting the execution of the system, or allowing the possibility for debugging.

We remark that the kernel module is fed *on-line* by events from the tracing subsystem. However, traces are not generated for off-line use, hence the module is not affected by the trace recording and storage overhead.

4.3 Implementation of SlackCheck

SLACKCHECK is a kernel module that implements the algorithm described in Section 3 as an automata using the kernel runtime verification subsystem. This module monitors the kernel scheduling events related to a specific task. The implemented automata is shown in Figure 3. From Definition 1, when the kernel module activates, it waits for a process of certain `PID` to have a `sched-in`, `sched-out` or `fork` event. When this event happens, the current system's timestamp is taken and refers to the out_0 value. At every following `sched-in` and `sched-out` event, slack is updated via the Equations (8) and (9) respectively, with the timestamps of the kernel `sched_switch` events taken as in_k and out_k. These events are gathered by placing a trace probe, executing SLACKCHECK's code before handling the event as normal, as per the Runtime Verification system, in Section 4.2.

At a `sched-in` event, the function in Listing 1 is ran. We first set in Line 6 the in_k value. If this is the first event found, then in Line 9 we assume that $\mathsf{out}_0 = \mathsf{in}_0$, and set both to the same timestamp. In Line 19, we update the slack value according to Eq. (8).

Listing 1 Slack update at a `sched-in` event.

```
1  static void rv_timed_update_in(u64 current_time)
2  {
3      struct rv_timed_struct *s = rv_timed_get_struct();
4      u64 diff = 0;
5  
6      s->ts_sched_in = current_time;
7  
8      if (!s->running) {
9          s->running = true;                  // out_0 = in_0 case
10         s->ts_sched_out = current_time;     // as per Eq. (7)
11         s->slack = s->delta;
12     } else {
13         if (s->ts_sched_in > s->ts_sched_out)
14             diff = s->ts_sched_in - s->ts_sched_out;
15         else                                // Sanity check: Out of order events.
16             diff = 0;                       // Avoid negative difference, we just
                  assume
17                                             // we missed an event.
18         if (diff <= s->slack)
19             s->slack = s->slack - diff;     // update of Eq. (8)
20         else
21             [handle process lower bound violation...]
22     }
23     return;
24 }
```

At a `sched-out` event, the function in Listing 2 is ran instead. We set in Line 7 the out_k value. If this is the first event found, it means the process was running before, and we start out monitoring of the lower bound from this point, taking the current time as our start point. Therefore we set in_{k-1} in Line 11, equal to the current time, entirely for precaution, as the value will be updated at the next `sched-in` event. We set the maximum slack possible as well in Line 13, as per Eq. (7). If instead this `sched-out` event was not the first, we run the update as normal in Line 22 as per Eq. (9).

The algorithm is translated into kernel code, showing the update of slack in the `sched-in` and `sched-out` events in Listings 1 and 2 respectively. This value is referred in the code as `s->slack`, while out_k and in_k are mapped to `s->ts_sched_out` and `s->ts_sched_in`.

The implementation was driven by the avoidance of floating point variables at kernel level. With this philosophy, slack is implemented as an integer, representing time in nanoseconds. The bandwidth α, however, cannot be represented as integer. Hence, we represent it as a rational number $\alpha = \frac{\alpha_{\mathsf{num}}}{\alpha_{\mathsf{den}}}$, with α_{num} and α_{den} being the numerator and the denominator respectively, stored in memory by the variables `s->alpha_num` and `s->alpha_den`.

Along with the listed code, other parts handle the passing of α and Δ parameters, along with handlers for the `fork` and `terminate` events. When the condition $\mathsf{slack} < 0$ happens in Listing 1, Line 21, the module forces an illegal automata transition, triggering the underlying reactor system, allowing the user to decide how to best handle the situation.

A sanity check, in Listing 1 (Line 15) and 2 (Line 17) respectively, controls whether SlackCheck receives events in the correct order. As a precaution we put a safeguard in both the `sched-in` and `sched-out` events. In case of an out of order event or the `current_time` is wrong, we assume the difference to be zero, and let slack stay the same.

The usage of our kernel module SlackCheck, is made through the runtime verification subsystem, using its file interface. The parameters are also set by writing to specific files, initially implemented in the `/proc` filesystem. The complexity is constant as the operations do not depend on the monitored task, and each update is also done in constant time. By writing the `PID` of the task or its process filename to the files `/proc/rv_timed_proc_id` and `/proc/rv_timed_proc_filename` respectively, the task starts to be monitored as soon as a scheduling events related to the task is detected.

Listing 2 Slack update at a `sched-out` event.

```
1  static void rv_timed_update_out(u64 current_time)
2  {
3      struct rv_timed_struct *s = rv_timed_get_struct();
4      u64 diff = 0;
5      s64 slack_diff = 0;
6
7      s->ts_sched_out = max(current_time, s->ts_sched_in);
8
9
10     if (!s->running) {
11         s->running = true;                     // out_0 case, as per Eq. (7)
12         s->ts_sched_in = current_time;
13         s->slack = s->delta;
14     } else {
15         if (s->ts_sched_out > s->ts_sched_in)
16             diff = s->ts_sched_out - s->ts_sched_in;
17         else                                   // Sanity check: Out of order evts
18             diff = 0;                          // Avoid negative diffrence, we
19                                                // just assume we missed an event.
20         diff = (s->alpha_den-s->alpha_num)*diff;
21         slack_diff = s->slack + (s64) div64_u64(diff, s->alpha_num);
22         s->slack = min_t(s64, s->delta, slack_diff);// update of Eq. (9)
23
24     }
25 }
```

The current implementation has an overhead due to attaching a probe for kernel scheduling events. This breakpoint adds a certain amount of latency that is evaluated in Section 5.3, that cannot be minimized. The number of operations is minimal in updating the slack value, at the cost of the precision. Since `slack` has been defined as an integer, we delay the division in Equation 9 as the last operation done, since integer division would discard the remainder value.

This discarded remainder from the integer division at Line 21 of Listing 2 may possibly build up some inaccuracy in the exact `slack` value. Since this value is the remainder of a division by α_{num}, assuming an uniform distribution of values, the average remainder would be $\frac{\alpha_{\mathsf{num}}}{2}$ nanoseconds from a `sched-out` update. Therefore the amount of inaccuracy is tied to how many `sched-out` events the process triggers. In this regard, we must underline that such accumulated error is reset as soon as it saturated by the upper limit of Δ at `sched-out` events, as indicated in Equation (9) and Line 22 of Listing 2. In the following experiments, we have empirically observed values in the order of 30 nsec for cases where `slack` did not become negative, and 180 nsec, when we allowed negative values, with $1 \leq \alpha_{num} \leq 40$. Since higher values of α_{num} have a more significant remainder, we recommend to enter the fraction representing α in its minimal terms.

Let us now evaluate possible incorrect results due to overflow. The timestamps are defined as a `u64` datatype (unsigned integers on 64 bits) for the monotonic clock used in the implementation. These timestamps are relative to the boot time, and SlackCheck also calculates the `slack` value by using relative timestamps. Therefore the value won't ever grow unbounded as time passes, as limited by the latency threshold Δ on a `sched-out` update due to the `min_t` function at Line 22 in Listing 2, except in case of negative slack on a `sched-in`, at Line 19 in Listing 1 which could also be modified to have a threshold value. A `u64` timestamp overflows after 584 years (or half of this time with the signed version), meaning that it is unlikely that it may happen. We remark that in the Linux kernel, the `u64` type for clock representation is ported even to architectures with a shorter word, defining the number of nanoseconds by a `long long` [9]. This would make the timekeeping slightly slower on 32 or 16 bits architecture as the datatype is emulated, but still correct.

[9] `https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/time_types.h#L9`

A slightly greater care must be taken at Line 20 of Listing 2, invoked at `sched-out`, where the multiplication `(s->alpha_den-s->alpha_num)*diff` may cause an integer overflow. Such an overflow, however, is very unlikely in reasonable circumstances. For example, if it is desired to express α with 3 digits, with $\alpha_{\mathsf{den}}1000$ and α_{num} prime with α_{den}, then the multiplication above would overflow if the task is continuously scheduled uninterruptedly for 213 days. Such a condition seems re-assuring that an overflow is very unlikely in realistic scenarios.

Finally, we would like to spend a few words on the instant of detection of a violation of the temporal constraint. In our implementation, such a violation is only detected at the next `sched-in` event, while it could be instead brought up earlier. In the example of Figure 2, the slack becomes negative already at time 14, whereas we do detect it at the next sched-in which is at time 15.

We can think of two possible alternate implementations:

- check at every scheduling event (not necessarily related to the task τ) or
- set a watchdog timer at every sched-out.

The first option of checking all events adds a latency and does not scale well with the number of tasks being monitored. The second option of the timer would require arming a timer at every sched-out, disarming at every sched-in, and implement a handler of the timer event.

5 Experiments

To test SlackCheck, we performed many experiments with the following common characteristics:

- CPU: `AMD Ryzen 5 3600`, with 6 cores.
- Fixed Clock Speed: 2.2GHz, as the minimum possible speed the CPU allowed for its scaling frequency.
- SMT/HT (Symmetric Multi-Threading/Hyper-Threading) disabled, through the `nosmt` kernel command line option, at boot time.
- Kernel Version: `v6.7`[10].
- SlackCheck's handling of negative slack values disabled, recording the entire process instead.

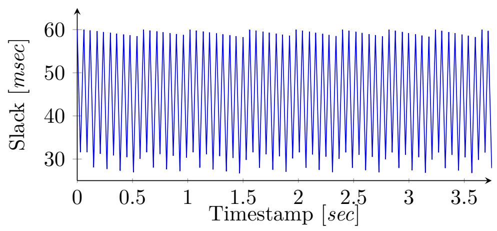
The choice of fixing the speed and disabling HT were made to avoid the uncontrolled interference of hardware mechanisms.

The workload was rendered by a multi-threaded application, in which each thread just keeps the CPU busy by counting up. The main thread is the task τ being monitored by SlackCheck. The others are used to represent the disturbances due to the external load. All the threads, both the monitored one and the interfering ones, are confined to execute over m cores, with m from 1 to 5.

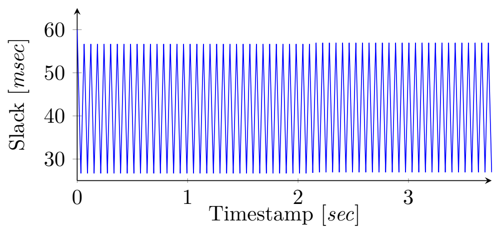
An experiment run is controlled by a shell script which does the following steps:

1. It sets a fixed frequency of the CPU clock;
2. It creates a `CPUSet`, a subset of available CPUs for the experiment threads.
3. It starts `dmesg` in order to gather SlackCheck messages.
4. It attaches the task to SlackCheck, also setting up its parameters α and Δ.
5. It starts all threads and enables the tracing kernel scheduling events with `trace-cmd` (to allow the visualization with KernelShark).
6. As the experiment is finished, it parses SlackCheck messages into a CSV file.

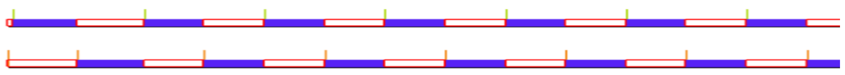
[10] As tagged in the Linux git repository at `https://git.kernel.org/`.



(a) Slack of the task when alone.



(b) Slack of the task when another on the same core.



(c) Portion of the schedule of the two tasks of the scenario of Fig. 4b as viewed by KernelShark.

Figure 4 Slack and schedule by `SCHED_DEADLINE`, with $m = 1$ core.

After the run, the traces of the scheduling events collected by `trace-cmd` are visualized by KernelShark[11]. This tool allows viewing kernel event traces, showing the timeline of processes receiving CPU time. In our KernelShark views, we represent the schedule of all threads created, with the monitored thread τ on top. Different colors represent the execution on different CPUs.

5.1 `SCHED_DEADLINE` experiments

As recalled earlier in Section 4.1, the class `SCHED_DEADLINE` schedules tasks by Global EDF, which is EDF allowing tasks to freely migrate among the cores in accordance to the affinity mask. Deadlines are used by `SCHED_DEADLINE` only to assign a dynamic priority to tasks and then schedule them. However, no deadline violation is reported by the kernel scheduler. Since we are testing a set of threads fully utilizing the m available cores, we disabled the `SCHED_DEADLINE`'s admission control.

In this experiment, we are using SlackCheck to experimentally validate the tardiness bounds for Global EDF [21, 44, 5]. Specifically, we borrow an example showing the tightness of a tardiness bound [5]. We pick a subset of m cores among the available ones. Our machine has a total of 6 cores, so we let m run from 1 to 5 to always reserve one core for other activities. We create $m + 1$ tasks, which never suspend themselves. These $m + 1$ tasks are confined to execute over m cores and are scheduled by `SCHED_DEADLINE` with

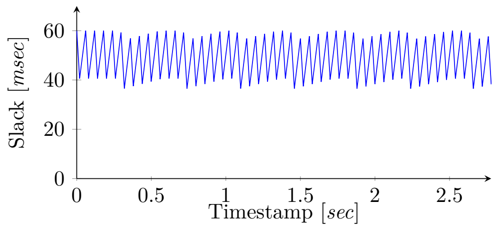
- a period P equal to 60 milliseconds, and
- budget $Q = \frac{m}{m+1}P$.

The choice of P is made to have no remainder when computing Q for m up to 5. The budget Q is chosen to have the bandwidth equal to $\alpha = \frac{Q}{P} = \frac{m}{m+1}$ and then have all m cores fully utilized.

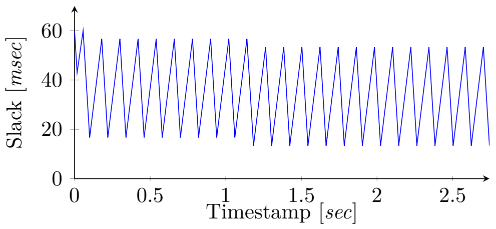
The delay Δ is set according to the implicit-deadline periodic server and accounts for the tight tardiness bound of this case [5], that is

$$\Delta = \overbrace{2(P-Q)}^{\substack{\text{standard } \Delta \text{ for}\\ \text{periodic servers}}} + \overbrace{\frac{m-1}{m+1}P}^{\substack{\text{tight tardiness}\\ \text{bound [5]}}} = \left(2\left(1-\frac{m}{m+1}\right)+\frac{m-1}{m+1}\right)P = P. \tag{16}$$

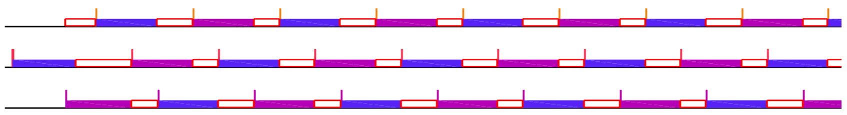
[11] `https://www.kernelshark.org/`



(a) Slack of the task when alone.

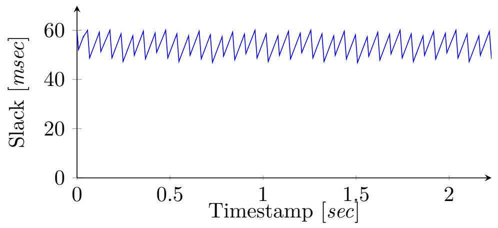


(b) Slack with other m tasks.

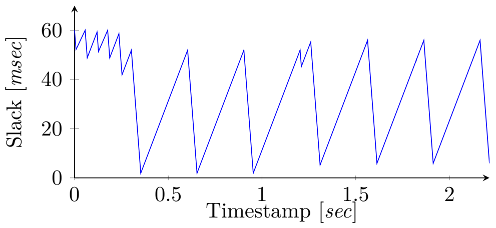


(c) Portion of the schedule for the scenario of Fig. 5b as viewed by KernelShark.

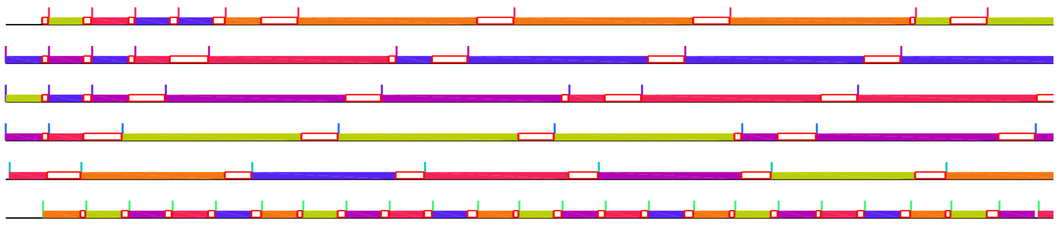
Figure 5 Slack and schedule by `SCHED_DEADLINE`, with $m = 2$ core. Compared to the case with $m = 1$ of Fig. 4, the task has less slack at sched-in instants.



(a) Slack of the task when alone.



(b) Slack with other m tasks.



(c) View of the schedule of the scenario in Figure 6b through KernelShark. Different colors mean to be scheduled over different cores. The first task is the one being monitored by SlackCheck.

Figure 6 Slack and schedule by `SCHED_DEADLINE` with $m = 5$ that is 6 tasks over 5 cores.

SlackCheck is then set up to check any violation of the slack from Theorem 2, with bandwidth α and delay Δ as set above.

Figure 4 shows the slack over time measured by SlackCheck, with $m = 1$ core only. In such a case, the task has a 50% utilization. Figure 4a shows the case where the task is alone in the system, whereas Fig. 4b shows the case with m extra tasks (only one in this case) with the same parameters, competing for the CPU. Figure 4c shows the schedule as viewed by KernelShark. In both cases, the slack shows a typical alternating behavior from a low value, at a sched-in instant, to a higher value, at the following sched-out instant. We observe that the scenario with zero slack is far from being reached. Such a scenario, in fact, appears when two idle intervals of length $P - Q$ are back-to-back, as shown in Figure 1. This schedule, however, despite being possible, did not appear in any of the experiments.

In Figure 5, we report the case with $m = 2$ cores. In this case, the reduction in the slack due to the presence of the other 2 tasks is more noticeable, as indicated by the lower values of the slack at sched-in instants in Figure 5b w.r.t. the values in Fig. 5a. In this case too, the schedule becomes very regular.

The cases with $m \in \{3, 4\}$ were run but did not reveal any additional insight. Hence, for brevity, we omit them. It is instead worth reporting and commenting the case with $m = 5$, shown in Figure 6. After a transient phase, the schedule becomes stable. Such a stable schedule repeats every $m \times P$, which is 300 milliseconds in this case. The slack of the task gets close to zero, confirming the validity of the value of Δ set by Eq. (16). Very few migrations occur (observed a few changes in the color of the schedule). Finally, we observe that the last task at the bottom of Fig. 6c is instead migrating at every job release and keeps being scheduled with the original period P.

5.2 SCHED_OTHER experiments

We also tested SlackCheck with `SCHED_OTHER`. The type of workload is the same described earlier, in which $m + 1$ threads always keep busy the m cores where they are confined to run to. With `SCHED_OTHER`, however, it is not possible to attach a precise CBS server with given budget and period. The `SCHED_OTHER` scheduling class tries to achieve fairness by implementing EEVDF [42].

Since we have $m + 1$ threads on m cores, we expect that a fair scheduler would assign a bandwidth of $\alpha = \frac{m}{m+1}$ to each of the thread. Hence, we set such a bandwidth value as the one to be verified by SlackCheck. The delay Δ, instead, depends on the scheduler internal frequency for updating scheduling decisions. To ease a comparison with the previous `SCHED_DEADLINE` case, we set it equal to the same value as Eq. (16), that is 60 milliseconds.

Figure 7 shows the slack of the task, when 2 same tasks are scheduled over a single core ($m = 1$). The scheduling decisions are made approximately every 3 msec. The slack shows a clear decreasing trend (highlighted by the red line in the figure). The slope of the trendline is -5.87×10^{-3} meaning that the actual bandwidth received by the task was not the theoretical value of 0.5 but about 0.494. Of course this is not surprising because `SCHED_OTHER` has lower priority than `SCHED_DEADLINE` and hence some of the bandwidth is necessarily assigned to other higher priority tasks.

Figures 8, 9, 10, and 11 shows again the slack of the task. From Equations (8) and (9, it follows that the slope of the slack should always be:

- -1 when the task is not scheduled
- $\frac{1-\alpha}{\alpha}$, which is $\frac{1}{m}$, when the task is scheduled.

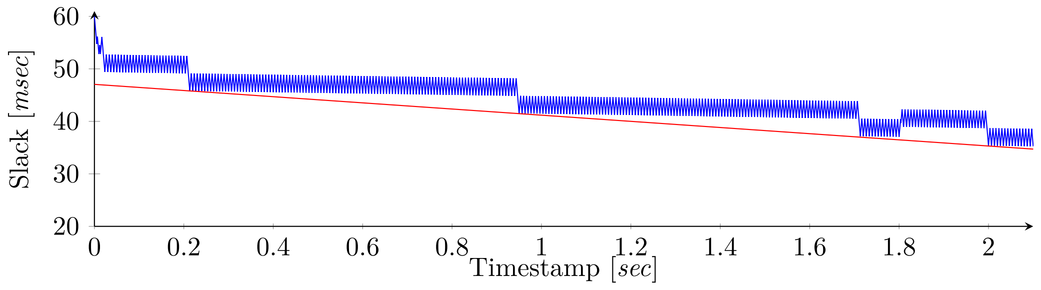


Figure 7 Slack by `SCHED_OTHER` with $m = 1$ that is 2 tasks over 1 cores. The slack oscillates at the frequency of scheduling decisions. The red trendline indicates that the constraint will be violated in the future.

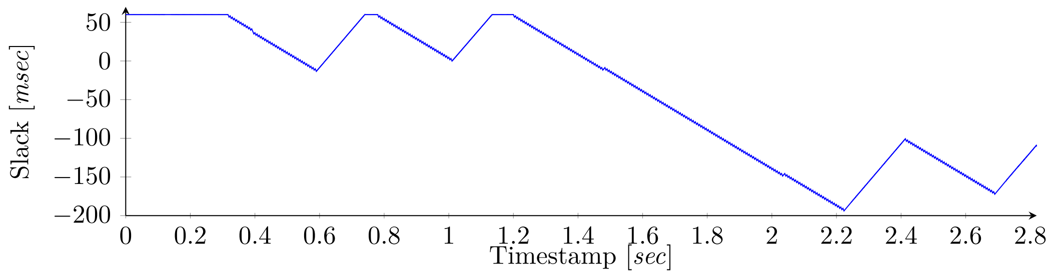


Figure 8 Slack by `SCHED_OTHER` with $m = 2$ (3 tasks over 2 cores).

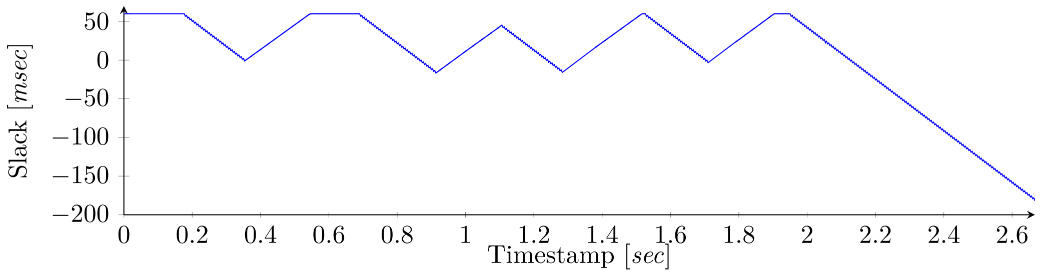


Figure 9 Slack by `SCHED_OTHER` with $m = 3$ (4 tasks over 3 cores).

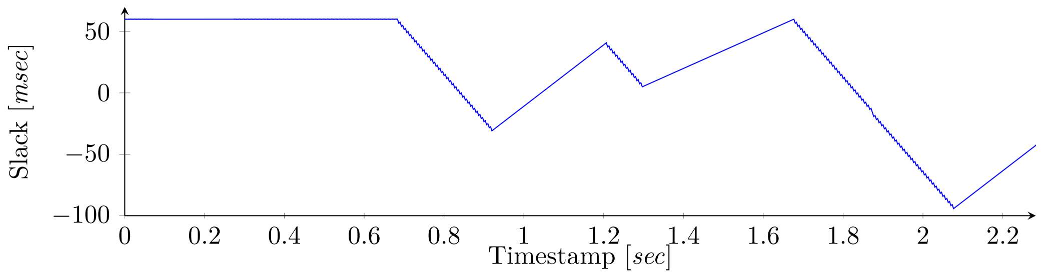


Figure 10 Slack by `SCHED_OTHER` with $m = 4$ (5 tasks over 4 cores).

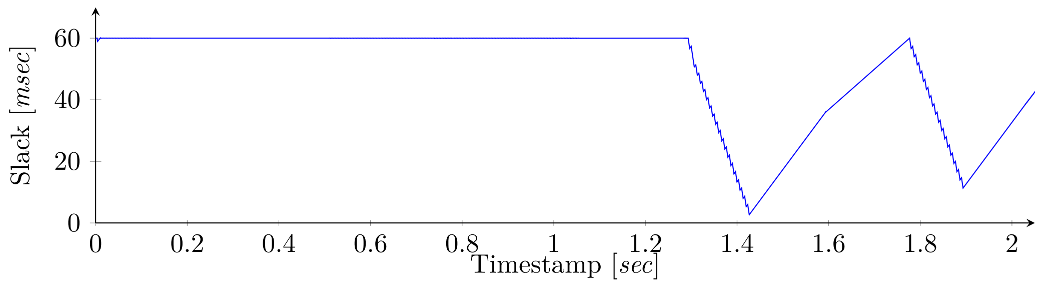


Figure 11 Slack by `SCHED_OTHER` with $m = 5$ (6 tasks over 5 cores).

We observe, however, some deviation from this ideal behavior, which we explain. In all figures, the segments of decreasing slack are due to a sequence of scheduling decisions in which the task receives alternately CPU time and idle time in **intervals of equal length**.

Table 1 Latency of SlackCheck measured by `cyclictest`.

	w/o stress		w/ stress	
	cyclictest	cyclictest + SlackCheck	cyclictest	cyclictest + SlackCheck
min [nsec]	48789	49270	54496	54314
max [nsec]	54681	53845	60151	72540
avg [nsec]	53252	53234	58410	58895

This results in an average decreasing slope of

$$\frac{-1 \text{ (when not scheduled)} + \frac{1}{m} \text{ (when scheduled)}}{2} = -\frac{1}{2}\left(1 - \frac{1}{m}\right).$$

For the segments where the slack increases, that is when the task is scheduled, the slack displayed could show a slope at a different rate. This is observed for the case in Figures 10 and 11. This different rate is due to update of the slack at some sched-out with the value of Δ, as required by Eq. (9). Therefore, a more correct representation, would have shown the same upwards rate throughout, hitting the threshold Δ before, and staying flat. Since SlackCheckonly updates the value at the *sched-in* the resulting line has a different slope.

The observed behavior is interesting. We recall, however, that our goal is not to delve into the Linux scheduler internals, but rather to present SlackCheck to verify temporal constraints expressed by a linear lower bound to the supply function.

5.3 Latency introduced by SlackCheck

In this section we describe the experiments to measure the latency of SlackCheck. For this purpose, we use the `cyclictest` program. This test measures the latency of a task that constantly requires CPU, but also leaves the resource as quickly, measuring therefore the latency of the system in giving back the resource to the program. For simulating a busy environment, and SlackCheck's worst case, we use `stress-ng` [12], with the `-switch` flag, introducing N stress threads, specifically overusing the system calls of `sched_yield`, in order to overwhelm the system with scheduling events, by constantly needing and releasing the CPU.

We run the cyclictest program for 50 runs, for 5 seconds each, as it performs around 5000 cycles of latency recording. For each run, we computed the average latency. Then, we computed maximum, minimum and average among all runs and we report them in Table 1.

The presence of cases with the latency with SlackCheck being smaller than the case without it, indicates that the latency is below the sensitivity of `cyclictest` and is dominated by other factors such as interrupts from devices, USB polling, or transferring of data. It is then confirmed that the impact of SlackCheck is minimal, as its constant time complexity indicates.

6 Conclusions and future works

In this paper, we have presented SlackCheck, a runtime verification engine which can detect the violation of temporal constraints expressed by a linear lower bound to the supply function. SlackCheck is implemented as a kernel module and has minimal overhead. Also, SlackCheck demonstrated its usefulness even in understanding the scheduler internals.

[12] `http://colinianking.github.io/stress-ng/`

The main future direction of investigation is related to the extension to the case of some internal parallelism of the "task" τ being monitored, possibly considering the parallel version of supply functions [39, 10, 29]. Other extensions related to the implementation of the module are about overcoming a single task limitation, as well as an eBPF version. This will allow a much greater freedom of usage, without the need for a kernel recompilation, and an easier interface for interacting with the system.

References

1 Benny Åkesson and Kees Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 851–856, 2011. doi:10.1109/DATE.2011.5763145.

2 Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, pages 4–13. IEEE, 1998. doi:10.1109/REAL.1998.739726.

3 Luca Abeni, Tommaso Cucinotta, Giuseppe Lipari, Luca Marzario, and Luigi Palopoli. Qos management through adaptive reservations. *Real-Time Systems*, 29:131–155, 2005. doi:10.1007/s11241-005-6882-0.

4 Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. 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), 24-27 September 2002, San Jose, CA, USA*, pages 133–142. IEEE, 2002. doi:10.1109/RTTAS.2002.1137388.

5 Shareef Ahmed and James H. Anderson. Tight tardiness bounds for pseudo-harmonic tasks under global-edf-like schedulers. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:24, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2021.11.

6 Luís Almeida and Paulo Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the* 4^{th} *ACM International Conference on Embedded Software*, pages 95–103, Pisa, Italy, September 2004. ACM. doi:10.1145/1017753.1017772.

7 Luís Almeida, Paulo Pedreiras, and José Alberto G. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Transaction on Industrial Electronics*, 49(6):1189–1201, December 2002. doi:10.1109/TIE.2002.804967.

8 François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and linearity*, volume 3. Wiley New York, 1992.

9 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, volume 10457, pages 1–33. Springer, 2018. doi:10.1007/978-3-319-75632-5_1.

10 Enrico Bini, Marko Bertogna, and Sanjoy Baruah. Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 437–446, Washinghton, DC, USA, December 2009. doi:10.1109/RTSS.2009.35.

11 Daniel Bristot de Oliveira, Daniel Casini, and Tommaso Cucinotta. Operating system noise in the linux kernel. *IEEE Transactions on Computers*, 72(1):196–207, 2023. doi:10.1109/TC.2022.3187351.

12 Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Efficient formal verification for the linux kernel. In *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*, volume 11724, pages 315–332. Springer, 2019. doi:10.1007/978-3-030-30446-1_17.

13 Artem Burmyakov, Enrico Bini, and Eduardo Tovar. Compositional multiprocessor scheduling: the GMPR interface. *Real-Time Systems*, 50(3):342–376, 2014. doi:10.1007/s11241-013-9199-8.

14 John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 111–126. IEEE, 2006. doi:10.1109/RTSS.2006.27.

15 Felipe Cerqueira and Björn Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *9th Annual workshop on operating systems platforms for embedded real-time applications*, pages 19–29. SYSGO AG, 2013.

16 Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1-2):25–53, 2002. doi:10.1023/A:1015394302429.

17 Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133, pages 5:1–5:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ECRTS.2019.5.

18 Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Proceedings of the 1962 spring joint computer conference, AFIPS 1962 (Spring), San Francisco, California, USA, May 1-3, 1962*, pages 335–344. ACM, 1962. doi:10.1145/1460833.1460871.

19 Rene L. Cruz. A calculus for network delay, part I: network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991. doi:10.1109/18.61109.

20 Tommaso Cucinotta, Dhaval Giani, Dario Faggioli, and Fabio Checconi. Providing performance guarantees to virtual machines using real-time scheduling. In *European Conference on Parallel Processing*, volume 6586, pages 657–664. Springer, 2010. doi:10.1007/978-3-642-21878-1_81.

21 UmaMaheswari C Devi and James H Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008. doi:10.1007/s11241-007-9042-1.

22 Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. An edf scheduling class for the linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop*, pages 1–8, 2009.

23 Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2–3):187–205, 2003. doi:10.1023/A:1025120124771.

24 Michael B Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. *ACM SIGOPS Operating Systems Review*, 31(5):198–211, 1997. doi:10.1145/268998.266689.

25 Paraskevas Karachatzis, Jan Ruh, and Silviu S Craciunas. An evaluation of time-triggered scheduling in the linux kernel. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems, RTNS 2023, Dortmund, Germany, June 7-8, 2023*, pages 119–131. ACM, 2023. doi:10.1145/3575757.3593660.

26 Neil Klingensmith and Suman Banerjee. Hermes: A real time hypervisor for mobile and iot systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications, HotMobile 2018, Tempe, AZ, USA, February 12-13, 2018*, pages 101–106. ACM, 2018. doi:10.1145/3177102.3177103.

27 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001. doi:10.1007/3-540-45318-0.

28 Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016. doi:10.1002/spe.2335.

29 Hennadiy Leontyev, Samarjit Chakraborty, and James H Anderson. Multiprocessor extensions to real-time calculus. *Real-Time Systems*, 47(6):562–617, 2011. doi:10.1007/s11241-011-9135-8.

30 Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems (ECRTS 2003), 2-4 July 2003, Porto, Portugal, Proceedings*, pages 151–158, Porto, Portugal, July 2003. doi:10.1109/EMRTS.2003.1212738.

31 Chenyang Lu, John A Stankovic, Sang H Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1-2):85–126, 2002. doi:10.1023/A:1015398403337.

32 Martina Maggio, Juri Lelli, and Enrico Bini. rt-muse: measuring real-time characteristics of execution platforms. *Real-Time Systems*, 53(6):857–885, November 2017. doi:10.1007/s11241-017-9284-5.

33 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, volume 9, 2009.

34 Aloysius K. Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS 2001), 30 May - 1 June 2001, Taipei, Taiwan*, pages 75–84, Taipei, Taiwan, May 2001. doi:10.1109/RTTAS.2001.929867.

35 Shuichi Oikawa and Ragunathan Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium, RTAS'99, Vancouver, British Columbia, Canada, June 2-4, 1999*, pages 111–120. IEEE, 1999. doi:10.1109/RTTAS.1999.777666.

36 Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 1(3):344–357, 1993. doi:10.1109/90.234856.

37 Rodolfo Pellizzoni, Bach Duy Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, pages 221–231. IEEE, 2008. doi:10.1109/RTSS.2008.42.

38 John Regehr and John A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, 2-6 December 2001*, pages 3–14, London, United Kingdom, 2001. doi:10.1109/REAL.2001.990591.

39 Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 181–190, Prague, Czech Republic, July 2008. doi:10.1109/ECRTS.2008.28.

40 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), 3-5 December 2003, Cancun, Mexico*, pages 2–13, Cancun, Mexico, December 2003. doi:10.1109/REAL.2003.1253249.

41 Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on networking*, 6(5):611–624, 1998. doi:10.1109/90.731196.

42 Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*, pages 288–299. IEEE, 1996. doi:10.1109/REAL.1996.563725.

43 Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems, ISCAS 2000, Emerging Technologies for the 21st Century, Geneva, Switzerland, 28-31 May 2000, Proceedings*, pages 101–104, 2000. doi:10.1109/ISCAS.2000.858698.

44 Paolo Valente. Using a lag-balance property to tighten tardiness bounds for global EDF. *Real-Time Systems*, 52(4):486–561, 2016. doi:10.1007/s11241-015-9237-9.

45 Y-C Wang and K-J Lin. Implementing a general real-time scheduling framework in the red-linux real-time kernel. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*, pages 246–255. IEEE, 1999. `doi:10.1109/REAL.1999.818850`.

46 Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in xen. In *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 27:1–27:10. ACM, 2014. `doi:10.1145/2656045.2656066`.

47 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, February 2016. `doi:10.1109/TC.2015.2425889`.

48 Peter Zijlstra. An update on real-time scheduling on linux (keynote talk). In *19th Euromicro Conference on Real-Time Systems*, 2017.