An Evaluation of Time-triggered Scheduling in the Linux Kernel

Paraskevas Karachatzis
TTTech Computertechnik AG
Vienna, Austria
paraskevas.karachatzis@tttech.com

Jan Ruh
TTTech Computertechnik AG
Vienna, Austria
jan.ruh@tttech.com

Silviu S. Craciunas
TTTech Computertechnik AG
Vienna, Austria
silviu.craciunas@tttech.com

ABSTRACT

The GNU/Linux operating system (OS) is becoming more commonly used in real-time systems and has been modified with specific updates to provide faster and bounded response times as well as with fixed- and dynamic-priority real-time scheduling mechanisms. While the EDF-based scheduler ensures deadlines and temporal isolation for real-time tasks, it may not provide the level of determinism needed for modern applications that also have to consider complex jitter and multi-rate task dependencies.

In this paper, we propose, implement, and evaluate an open-source, kernel-level time-triggered scheduling approach for Linux, examining the level of determinism achievable in terms of task execution and end-to-end latencies. We show that time-triggered scheduling in the Linux kernel achieves reduced latency and jitter for real-time applications when compared to the existing scheduling policies and user-space time-triggered implementations. Additionally, in terms of end-to-end communication latencies for distributed real-time applications, we compare a software-based IEEE 802.1Qvb time-aware gating implementation for time-sensitive networking (TSN) in which the time-triggered application schedule can be aligned to the network schedule to the standard Linux networking subsystem.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**.

KEYWORDS

Time-triggered scheduling, Linux kernel, Time-sensitive Networks.

ACM Reference Format:
Paraskevas Karachatzis, Jan Ruh, and Silviu S. Craciunas. 2023. An Evaluation of Time-triggered Scheduling in the Linux Kernel. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 7–8, 2023, Dortmund, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3575757.3593660

ACKNOWLEDGMENTS

This paper is supported by European Union's Horizon Research and Innovation Programme under Grant Agreement number 101076754, Project AITHENA.

RTNS 2023, June 7–8, 2023, Dortmund, Germany

1 INTRODUCTION

The GNU/Linux operating systems (OS) were not envisioned and designed for use in real-time application domains but are used as general-purpose OS capable of running on both embedded targets as well as desktop and server systems [52]. However, in many domains such as industrial control, Internet of Things (IoT), and automotive systems, GNU/Linux has become an attractive candidate for use in (soft) real-time applications due to the emergence of mixed-criticality workloads [35], the potential for cost reduction through COTS usage [50], the attractiveness of open-source licensing models, and the need for extensive library support and well-established programming environments [50]. This evolution manifests itself in the increased interest in academia and industry to comprehend and enhance the Linux kernel's real-time capabilities [19]. Nowadays, the Linux community has extended the kernel to provide faster and bounded response times (through the PREEMPT_RT patch) [50] and introduced fixed- and dynamic-priority real-time scheduling in the form of the SCHED_FIFO and SCHED_DEADLINE schedulers. SCHED_DEADLINE [39] implements Constant-Bandwidth Server (CBS) scheduling [2] that is a variant of Earliest-Deadline First (EDF) also featuring temporal isolation via CPU reservations. A survey of real-time enhancements in the Linux Kernel can be found in [60], and a discussion on the remaining challenges within the throughput-oriented design of Linux can be found in [43].

While the EDF-based real-time scheduler SCHED_DEADLINE will ensure temporal isolation and deadlines of real-time tasks, it may not offer the strict determinism needed in modern applications that goes beyond meeting deadlines and also needs to consider more complex timing dependencies. For example, in the automotive domain, there has been a shift towards a more centralized functionality featuring scalable and flexible integrated hardware platforms (c.f. [48]) that enables complex real-time requirements of, e.g., Advanced Driver-Assistance Systems (ADAS)/Autonomous Driving (AD) [22, 44]. In particular, it is challenging to support the non-trivial jitter requirements and multi-rate dependency chains necessary in ADAS/AD functions [6, 7] using EDF scheduling. Such applications benefit from a more predictable, time-triggered (TT) execution [44, 56]. Time-triggered scheduling has proven superior in terms of determinism, stability, predictability, and compositionality [21, 41, 45, 51, 61, 62], and thus has been successfully deployed in real-world applications. For example, MotionWise [57, 58] is a middleware layer running on top of GNU/Linux and proprietary (real-time) OSs that enables time-triggered scheduling independent of the underlying OS scheduling [45] by implementing user-space time-triggered scheduling. However, user-space scheduling exhibits higher latencies with more variations (c.f. Sect. 3.3).

Deterministic task execution is a necessary but insufficient condition to fulfill modern applications' real-time requirements. Most

modern applications communicate with each other, thus also requiring determinism on the networking part [5] and synchronization between the application execution and the message transmission over the network. However, the Linux kernel's networking subsystem has been designed with throughput rather than determinism in mind. Hence, we also need to explore how to add real-time capabilities to the communication subsystems. Recently, Time-Sensitive Networking (TSN) [29] has introduced a set of standardized mechanisms and protocols enhancing IEEE 802.1Q [27] bridges with real-time capabilities such as clock synchronization (IEEE 802.1AS-rev [30]) and time-aware shapers (IEEE 8021.Qbv [28]) enabling a global time-triggered communication schedule [18, 53, 59]. On the end-system nodes, support for real-time communication via TSN has been added, e.g., in the Time-Aware Priority Shaper (TAPRIO) project [32]. TAPRIO adds the IEEE 802.1Qbv time-aware shaping to the Linux communication subsystem implementing egress queues that can be configured as a timed sequence of open or closed gate states, enabling or disabling frame transmission based on a schedule defined in so-called Gate-Control Lists [38].

In this paper, we implement and evaluate an open-source time-triggered scheduling and communication approach for Linux, studying the degree of determinism, both in task execution and communication latencies, that can be achieved using this paradigm. We show that a time-triggered scheduler (SCHED_TT) fits seamlessly into the Linux kernel scheduling hierarchy, but certain aspects related to task activation, thread spawning, and synchronization need to be done carefully. Furthermore, we investigate the benefits of the implementation concerning its reduced variance in execution and communication latencies for real-time applications when compared to existing scheduling policies and a user-space time-triggered implementation. For the communication latencies, we compare the TAPRIO queueing discipline (qdisc) (software-based IEEE 802.1Qvb time-aware gating) [32] against the standard Linux networking subsystem. The main contributions of our paper are the implementation of the open-source kernel-level SCHED_TT scheduler and an extensive evaluation showing the benefits and trade-offs of adding the time-triggered paradigm to the Linux kernel.

We survey related works on real-time Linux in Section 2, followed by a description of our time-triggered scheduling implementation in Section 3. We then present an extensive experimental evaluation in Section 4 and conclude the paper in Section 5.

2 RELATED WORK

Linux has been used in a series of real-time, from robotics applications and industrial systems, to fog and edge computing [25, 39, 52]. One of the first serious efforts to enhance the real-time behavior of Linux was the PREEMPT_RT patch [50]. The SCHED_RT policy implements fixed-priority scheduling via 99 priority levels for real-time tasks. However, fixed priority scheduling, as is implemented now, does not offer protection (in the form of temporal isolation) against starvation when a high-priority task exceeds its worst-case execution time (WCET) assumption. In order to alleviate this problem and introduce the concept of deadlines to the Linux kernel, the SCHED_DEADLINE scheduler was introduced, implementing an EDF-variant with temporal isolation and slack reclaiming. A survey of real-time enhancements in the Linux kernel can be found in [60]. Additionally, many projects have implemented Linux variants with some type of real-time support [20, 23, 25, 42]. The LInux Testbed for MUltiprocessor Scheduling in Real-Time Systems (LITMUS^{RT}) project [13] is an experimental extension to the Linux kernel that provides abstractions and interfaces that ease the integration of real-time scheduling and synchronization algorithms for multiprocessor systems. Currently, LITMUS^{RT} implements the partitioned, global, and clustered EDF (PSN-EDF, GSN-EDF, C-EDF), partitioned Fixed-Priority (P-FP), Partitioned Reservation-Based Scheduling (P-RES), and pfair (PD^2) algorithms. Additionally, LITMUS^{RT} also supports table-driven scheduling based on ARINC 653 partitions where each scheduling slot in the static schedule table can reference a set of processes that are then dispatched dynamically within the slot. LITMUS^{RT} was compared to vanilla Linux and Linux with the PREEMPT_RT in terms of scheduling latency in [15].

In the automotive domain, a typical OS that is used is AUTOSAR, which has a fixed-priority dispatcher with the option to put offsets on runnables which can be seen as a restricted version of table-driven scheduling [26]. Alternatively, to achieve more deterministic behavior, time-triggered scheduling has been introduced as a user space feature via the MotionWise solution [57, 58]. MotionWise emulates a time-triggered execution via standard POSIX system calls and forces the execution of the tasks according to the schedule table. While user-space scheduling is faster to develop, easier to debug, and more portable (e.g., to other POSIX systems), it has higher latencies and more runtime variance that may degrade the desired strict determinism. Implementing the scheduler within the kernel has the advantage of removing traps and direct control when context switching, reduced latencies due to fewer system calls crossing the user-space kernel-space boundary, and access to the physical memory (and, in general, other hardware resources). On the other hand, there are some disadvantages to a kernel implementation primarily related to stability (bugs may crash the kernel), maintenance (dependence on Linux scheduler classes), and portability (new kernels require modifications or even implementation from scratch and merging changes upstream).

Some studies have investigated the degree of determinism regarding real-time latencies for Linux. Amongst these, [4] measured latencies in the Linux kernel via micro-benchmarks, showing that between the timer resolution and non-preemptable sections, the timer resolution latency is dominant. Additionally, in [19], the authors study the scheduling latency, including all possible synchronization flows in the PREEMPT_RT Linux Kernel, introduce a tracing tool for kernel events, and show that preemption and sections with disabled interrupts add most to the scheduling latency.

In terms of determinism beyond the temporal isolation offered by SCHED_DEADLINE, most works focus either on spatial isolation or reducing cache interference. While containers only offer spatial isolation, there have been efforts to introduce real-time containers through the implementation of hierarchical scheduling [1]. The spatial isolation property offered by containers is orthogonal to the desired temporal properties offered by time-triggered scheduling. Moreover, cache interference reduction (either via software or hardware support) [16, 36] is also orthogonal to our problem since it serves to reduce the pessimism of the WCET bound used in the generation of the offline schedule tables.

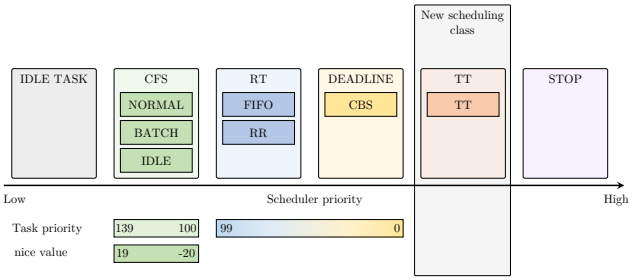

Figure 1: The Linux kernel's scheduling structure with the new TT scheduling class.

In terms of communication latencies, some studies [14, 24] investigate the applicability of the Linux kernel's communication stack (either unmodified or via open-source EtherCAT alternatives) to real-time systems. Time-Sensitive Networking (TSN) is a standardized alternative for real-time communication in some domains like industrial automation and automotive. In GNU/Linux, the TAPRIO project adds support for the timed-gate mechanism defined in IEEE 802.1Qbv [28] within the communication subsystem. The work in [38] investigates TAPRIO performance and latency in different re-routing scenarios. We also use TAPRIO and study its ability to improve communication latencies and enable an alignment between the time-triggered schedule of the sender and receiver applications with the schedule of the communication between them.

3 LINUX TT SCHEDULER IMPLEMENTATION

The Linux kernel recognizes real-time and regular (non-real-time) tasks, where the goal for regular tasks is to improve throughput/-fairness, and the goal for real-time tasks is to improve latency and maintain deadlines. Since Linux kernel v2.6.23, the scheduling subsystem of the Linux kernel has been revamped to be easily extendable with new and custom scheduling classes and policies [39]. We will now briefly describe the Linux kernel's scheduling classes for regular and real-time tasks to then proceed with the details of our new time-triggered scheduler class implementation.

3.1 Linux Kernel Scheduling Policies

The Linux Kernel provides three scheduling classes and five schedulers, organized in a hierarchy as depicted in Fig. 1. All schedulers are called in a well-defined order: (1) stop_sched (a pseudo-scheduler used for certain kernel threads), (2) SCHED_DEADLINE (the CBS scheduler), (3) the RT scheduler (SCHED_FIFO and SCHED_RR which are fixed-priority schedulers), (4) the CFS completely fair scheduler, (5) the IDLE scheduler [19]. In addition, our new TT scheduling class (depicted in red) is added to the scheduling hierarchy with a higher priority than the deadline CBS scheduler. Each time the scheduler gets invoked, the highest-priority scheduling class containing ready tasks is called. If there are no ready tasks in any of the scheduling classes, the special IDLE TASK scheduler returns an idle thread that is always ready to execute.

3.1.1 Completely Fair Scheduler (CFS). The completely fair scheduler (CFS) has been the standard scheduler for desktop tasks since its introduction to the Linux kernel v2.6.23. CFS implements the weighted fair queuing policy [49] but also provides some prioritization via the nice value which is essentially a priority field. CFS attempts to emulate the fairness of generalized processor sharing (an ideal policy in which the time is infinitely divisible) through the concept of "virtual runtime". For more information on CFS we refer the reader to [17, p.308ff].

3.1.2 Fixed Priority Scheduler (FiFo/RR). The Linux kernel implements a POSIX-compliant fixed-priority real-time scheduler (RT) with 99 priority levels which executes tasks in order of decreasing priority [15]. The RT scheduler has two different policies when two or more tasks have the same priorities. SCHED_FIFO selects equal-priority tasks according to their enqueuing order, while SCHED_RR implements a round-robin policy. With SCHED_FIFO, there is no maximum time slice used to preempt a running task; hence, an executing task will only be preempted by an incoming I/O request, the arrival of a task with higher priority, or if it voluntarily calls the yield function. SCHED_RR has a higher priority than SCHED_FIFO and imposes a maximum quantum for task execution, resulting in equal (highest) priority tasks being able to preempt each other.

The RT scheduler has priority over the CFS scheduler, i.e., tasks assigned to this scheduling class will take precedence over tasks assigned to the CFS class, and can be used to implement, e.g., rate-monotonic (RM) and deadline-monotonic (DM) policies [54]. Even though the RT scheduler can execute tasks such that they finish before their deadlines, if the appropriate schedulability tests hold, there are certain trade-offs comparing to, e.g., Earliest-Deadline First (EDF) [10, 12]. Primarily, fixed-priority schedulers do not provide temporal isolation amongst tasks resulting in starvation of all real-time tasks if, for e.g., a task is set to the highest priority and misbehaves in that it does not relinquish control over the CPU [52]. Moreover, the priority inheritance implementation may result in some deadline misses under certain conditions.

3.1.3 Deadline Scheduler (DL). The Deadline scheduler introduced with Linux kernel v4.13 is based on the Constant-Bandwidth Server (CBS) [2, 3] algorithm, which is an extension of EDF supporting temporal isolation (via resource reservations) and providing improved response times for aperiodic tasks. Like other server mechanisms, a CBS is characterized by a period and a budget but also has a dynamic deadline that is recomputed according to a set of rules (c.f. [3] for an in-depth description). If periodic tasks are encapsulated within their own CBS, a misbehaving task will not influence the temporal guarantees of any other real-time tasks. Moreover, because CBS is work conserving, overflows from real-time tasks can be mitigated by using slack made available by other tasks finishing earlier than their WCET assumption. SCHED_DEADLINE has higher priority than the RT scheduler in the Linux scheduling hierarchy.

While the CBS approach is superior to the RT policy due to the temporal isolation property, it still cannot easily handle the determinism required in some modern complex applications, e.g., regarding execution jitter and multi-rate cause-effect chains. This is mainly due to allowing jitter in the runtime execution which may result in breaking individual job-level dependencies. Although it has certain downsides (like being inflexible), time-triggered scheduling has advantages for such types of applications. Time-triggered scheduling does not allow any variance at runtime, and hence, it will not break any job-level dependencies at runtime [61].

3.2 Time-Triggered Scheduler (TT)

Time-triggered scheduling has multiple advantages for highly critical systems [21, 40, 41, 45, 51, 61, 62]:

- *Temporal isolation*: it supports correct-by-design approaches with strict temporal isolation between tasks,
- *Complex timing requirements*: it is able to satisfy complex constraints like the cause-effect chains from ADAS systems [44] as well as different types of jitter requirements,
- *Schedulability*: it supports higher system utilization and has a higher solution space, especially for complex task sets, compared to e.g. fixed-priority scheduling (c.f. Sec.3.2 of [62]).
- *Determinism*: it features increased stability and testability, and TT systems do not have unwanted run-time effects making the system behavior easier to analyze,
- *Synchronization to communication*: it is easy to achieve stable real-time behavior of cause-effect chains across the network domain,
- *Predictability*: many system properties become predictable, e.g., locks and task preemption,
- *Compositionality*: it supports compositional design and incremental scheduling, making system integration of SWCs from different suppliers easier.

A major downside of time-triggered scheduling is the lack of flexibility to adapt to runtime changes. Moreover, creating time-triggered schedules based on WCET estimates can lead to resource overprovisioning since the upper bound on the WCET is usually overly pessimistic. While the WCET pessimism is also a problem for any scheduling algorithm that offers design-time guarantees, in TT systems it can lead to a low resource utilization. Typically resource utilization is improved by allowing lower priority tasks to reclaim the unused time in TT slots (like in our approach) or by using average execution time estimates and employ more flexibility through methods like slot-shifting [33].

3.2.1 SCHED_TT Implementation. We have developed an open-source[1] time-triggered Linux kernel scheduler to enable executing tasks according to a statically-defined schedule table (under kernel version 5.9.1 with PREEMPT_RT). We added the TT scheduler to the Linux kernel's scheduler hierarchy with a higher priority than the deadline scheduler (c.f. Fig. 1) to ensure that no other task can block TT tasks as they execute based on the cyclic offline schedule. In the idle slots of the static schedule or when a TT task has finished earlier than its provisioned slot, tasks belonging to other scheduling classes will execute.

The user can specify a static schedule table per core (partitioned approach) that the respective core repeats every cylce or hyper-period. Internally, we represent the schedule table by a linked list, so picking the next task to run has complexity O(1). Furthermore, we added clock synchronization to the scheduler such that there is a global time reference for distributed systems that can enforce synchronized task execution and communication across hosts. The TT scheduler's timers run based on the POSIX clock CLOCK_MONOTONIC to avoid unexpected time jumps when the system synchronizes to an external clock source (e.g., network time). Synchronization to an external clock is done via one or more

[1]Will be made available at https://github.com/tttech-group

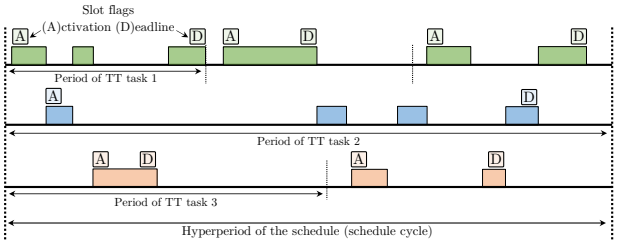

Figure 2: Example TT schedule with activation and deadline flags for slots.

dedicated synchronization slots in the TT schedule, which are reserved for state correction. During these synchronization slots, the scheduler adjusts its alignment to the external clock by either enlarging or reducing the size of the synchronization slot within its reserved bounds. Thus, the schedule can be aligned with an external clock source without negatively influencing the execution of critical tasks. Moreover, there is a trade-off when selecting how many synchronization slots to use and where to place them; the more synchronization slots there are, the more accurate and faster the synchronization becomes at the expense of solution space for fitting TT tasks in the static schedule table.

The static schedule table (c.f. Fig. 2 for an example) is created offline via heuristics or exact methods (e.g. [46]). When creating the schedule, the system designer defines a macrotick *mt* (also sometimes referred to as *slot length*), which captures the granularity of the schedule slots within the table [34]. A static schedule maps tasks to time slots, selecting one (or no) task to run in each interval. The schedule has a hyperperiod, after which the schedule repeats. Each slot in the schedule table has an additional flag that defines whether it is an activation (A) and/or deadline (D) slot (c.f. Fig. 2). If a slot is marked as an activation slot, the corresponding task is released at the beginning of the slot. The definition of activation slots prevents tasks that finish execution before their WCET to execute twice per period. Conversely, if a slot is marked as a deadline slot, the scheduler checks if the task has finished execution before the end of the respective slot.

The time-triggered scheduler uses high-resolution timers to start and stop tasks. The scheduling subsystem in Linux provides several standard function APIs for implementing any new scheduler. One of these functions is `enqueue_task()`, which adds a task to the standard ready-to-pick list when it is ready, i.e., released via the corresponding schedule slot in the schedule table. A task may not be ready at the start of its activation slot, but `enqeue_task()` will add it to the next list if it becomes ready at any time within the slot and mark the core for rescheduling. As a result, lower-priority background tasks cannot consume execution time from TT tasks that may wake up later in their slots (e.g., due to blocking semaphores). When a reschedule event is triggered, either due to the beginning of a slot in the TT schedule or a TT task becoming ready within its slot, a timer interrupt is set for the end of the current time slot. The activation of the timer interrupt marks the end of a slot so that the TT scheduler proceeds with the next time slot.

Furthermore, the timer interrupt indicates if a task has exceeded its deadline, given that the current slot is marked with the (D)eadline

flag. If the task currently running finishes execution (i.e., it is no longer in the RUNNING or READY state) before the end of the slot, the function `dequeue_task()` will be called canceling any active timers, removing the task from the ready-to-pick list and marking the core for rescheduling. In this way, any lower-priority background tasks may reuse leftover time from a TT slot. As usual in the Linux scheduling subsystem, we note that the scheduler timer does not perform the context switch but only signals that a context switch is pending. Instead, the Linux kernel performs the context switch when there is an explicit call to the `schedule()` function or when exiting an interrupt service routine. Moreover, we can request rescheduling by programming a timer interrupt. For example, the deadline scheduler uses this to preempt tasks that have passed their deadlines. We also use this method in the time-triggered class since the execution of tasks is driven by time and not events. When such a timer interrupt happens, we can immediately reschedule and keep the latency low. Finally, we emphasize that the reschedule function does not interfere with the static schedule table. However, it allows tasks of lower priority scheduler to exploit unused time from TT tasks.

When rescheduling, the `schedule()` function calls the associated `pick_next_task()` implementation for each scheduling class in order of their priorities to get the next task to execute. If there is no ready task in the list, the `schedule()` picks the next scheduling class in the scheduler hierarchy until `pick_next_task()` returns a ready task or the idle task. Hence, if the TT `pick_next_task()` returns without a ready task, e.g., because the task completed execution early, tasks from other scheduling classes utilize the remaining time slot.

Finally, we note that several functions of the Linux kernel scheduler API do not apply to SCHED_TT and have not been implemented or return zero. For example, `balance()` and `migrate_task_rq()` are not used by the time-triggered scheduler at the moment since each task is pinned to a specific core (partitioned approach) to eliminate migration overhead. However, in a semi-partitioned approach [9] that we leave for future work, these functions may be needed.

3.2.2 Time-Triggered Tasks. Real-time tasks require wrapper code to either enforce a periodic execution, prevent starvation, or adapt to the underlying scheduler [8]. Similarly, for tasks to run under the SCHED_TT policy, they must follow a specific structure. In Alg. 1, we show the pseudocode implementation of a TT task using the code from [8] as a template but modifying it as follows. Firstly, the `init()` function performs optional initialization steps, e.g., memory allocation, setting up sockets, or initializing variables. We note that at this point, the task is running with the inherited parent priority. Once initialization has completed, the TT task must call `set_tt_sched()` to switch the scheduling policy to SCHED_TT. The TT task wrapper then enters a loop and calls `sched_yield()` to wait for SCHED_TT to activate a job in the next reserved activation slot of the static schedule table. After the call to `sched_yield()`, we place the real-time payload of the task. At the task's next activation slot, SCHED_TT resets the activation flag, `sched_yield()` returns, and the payload of the task starts execution. When a TT job completes, it yields by calling `sched_yield()` and waits for its next activation slot. If a TT job fails to yield until the end of its (D)eadline slot, SCHED_TT preempts the task and records the deadline miss. At the moment, we apply a recovery policy for tasks that missed their deadline so that they resume their execution with their next activation slot. However, other strategies are possible and easy to configure. For example, an affected task may be restarted so that it is aligned again at the next activation slot, or it can be terminated if it is considered unsafe to resume execution.

Finally, unlike wrappers for periodic tasks scheduled using SCHED_RT [8], we do not need to keep track of user space timers to enforce periodic execution, since SCHED_TT inherently dispatches jobs periodically as part of the logic executed in activation and deadline slots associated with the TT tasks. Hence, the responsibility for periodic task execution is shifted from user space to the kernel scheduler. We also note that the wrapper code is not generally necessary for TT tasks to run in their reserved slots and to ensure temporal isolation. Any task can be configured as a TT task, and it will be executed in its reserved time slots. However, in this case, SCHED_TT cannot take into account the activation and deadline slots so that it cannot guarantee deadlines or periodic activation.

Algorithm 1 TT task featuring wrapper code

```
1: procedure MAIN
2:     init()
3:     set_tt_sched()
4:     while true do
5:         sched_yield()
6:         PAYLOAD
```

3.2.3 TT Task Pools. Note that forking TT tasks is problematic, as it requires child processes to share the existing time slots with their parent. We can account for this during schedule generation and refer to the resulting special time slot as a TT pool that internally is represented by a tgroup (in reference to cgroups in Linux containers). Tasks co-located in the same TT pool are scheduled inside the pool according to a pool-level scheduling policy. For this early experimental version of TT pools, we implemented a FIFO approach for scheduling tasks within the pool, but other policies, such as (weighted) round-robin or priority-based, can be easily added. We want to point out, that our functionality of TT pools closely reassembles LITMUS^{RT}s table-driven reservations [13]. However, TT pools are only an experimental functionality and not the main focus of our paper. Notably, table-driven reservations or TT pools could be used to schedule Linux containers as black boxes. For the current work we only utilize containers for their spatial isolation properties with a single TT task per container.

3.3 User vs. Kernel-space TT scheduling

As described in Section 2, time-triggered scheduling has been implemented for Linux (and other POSIX-compliant systems) in user space. One prominent example that is used in several real-world automotive projects is MotionWise [58], which is, for example, part of the piloted driving platform of the Audi A8 [31]. MotionWise emulates a time-triggered execution in user space via POSIX-compliant system calls to the kernel. While this approach certainly has benefits (being faster to develop, easier to debug, and more portable to

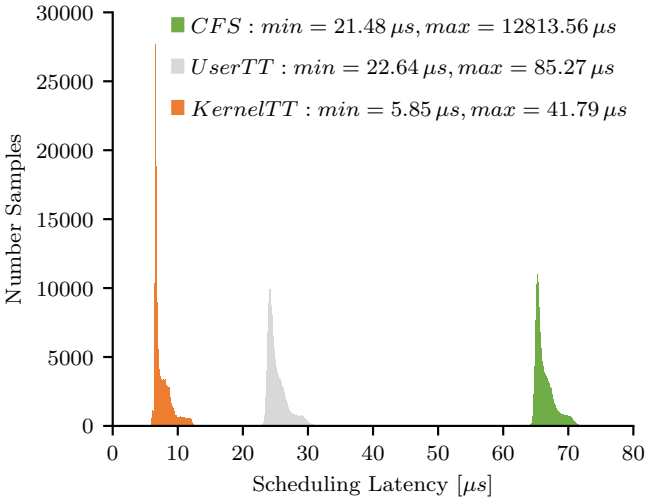

Figure 3: A comparison of user space and kernel space TT schedulers in terms of task response-times and jitter

.

other POSIX systems), the latencies experienced by real-time tasks are significantly higher and have more runtime variance.

We show an experiment comparing our SCHED_TT kernel-level scheduler with a time-triggered scheduling implementation in user space, similar to the MotionWise middleware [58]. We show a histogram of task response times in Fig. 3, also comparing to CFS as a non-real-time baseline. We see that while the user-space implementation is significantly more deterministic than CFS, the tasks still have a higher response time and a higher degree of variability in response times, as well as some outliers. Our kernel-level implementation, on the other hand, exhibits the least overhead, resulting in lower response times with less variance for tasks. We note that there are significant outliers for CFS (with a maximum of 12.8ms) that are not shown in Fig. 3 to better visualize the differences between the kernel and user-space variants. The mean and standard deviation (mean, std) of the task response times for CFS, user-space TT, and kernel-space TT are $(67.4\mu s, 42.7\mu s)$, $(25.5\mu s, 2.8\mu s)$, and $(7.7\mu s, 1.7\mu s)$, respectively.

We will now present a more comprehensive suite of experiments to show the potential of kernel-level TT scheduling for distributed applications with strict determinism and low jitter requirements, comparing also to the other real-time scheduling policies and showing the limitations of the TSN-based real-time network subsystem.

4 EXPERIMENTAL EVALUATION

We evaluated the Linux kernel's real-time capabilities using the SCHED_FIFO scheduler, the SCHED_DEADLINE scheduler, and our SCHED_TT scheduler by measuring the end-to-end latency between two communicating containers facing interfering workloads with three different Linux bridge configurations. The end-to-end latency represents the duration elapsed from the start of execution of the sender until the end of the execution of the receiver tasks. Moreover, in an additional setup we compare SCHED_TT against the LITMUS^{RT} table-driven scheduler.

When assessing the end-to-end latency, we are not interested in its absolute value but in the degree of jitter, which is the difference between the maximum and minimum measured latency. As such, the end-to-end latency jitter is a measure of timing determinism. The degree of timing determinism achievable with the Linux kernel depends strongly on the configuration of various kernel parameters and the setup and usage of features such as the Linux kernel's queueing disciplines (qdiscs) and the scheduling policy. Therefore, we performed our experiments with a fixed base configuration that minimized the system latency as a component of the end-to-end latency and varied the configuration concerning the kernel scheduling policy (SCHED_FIFO, SCHED_DEADLINE, SCHED_TT) and the Linux software bridge. In the following, we describe our experimental setup, including the Linux kernel configuration, the Linux bridge configuration, and the workloads represented by four containers and a set of native real-time tasks.

4.1 Experimental Setup

We visualize the experimental setup for our experiments in Fig. 4.

4.1.1 Linux Host Configuration. For our experiments, we used an edge computing device with an Intel Atom E3950 with four cores at 1.594 GHz each and 8 GiB of main memory running Ubuntu server 22.04.1. For SCHED_TT experiments we ran Linux kernel v5.9.1. For SCHED_DEADLINE the vanilla Linux kernel v5.9.1 would crash with a kernel panic in the priority inheritance code. As a result, we opted to use Linux kernel v6.2.0 for SCHED_DEADLINE and SCHED_FIFO experiments. We ran the experiments with PREEMPT_RT, disabled RT-throttling, no hyper-threading, disabled CPU power management (C-states and p-states), set the kernel to use the scaling governor to maximize the CPU frequency, and raised the priority of interrupt threads to SCHED_FIFO priority 50. In the following, we elaborate on the specific configurations:

PREEMPT_RT Using the PREEMPT_RT Linux patch enables big portions of the Linux kernel code to become preemptible, including interrupt handlers, reducing system latency of high-priority tasks. Notably, interrupt handlers are split into an atomic low-level interrupt handler that is non-preemptible and a preemptable high-level handler executing as a kernel thread.
Disabling RT-Throttling RT-Throttling prevents high-priority tasks, scheduled with the FiFo policy, to introduce long latencies due to programming faults acting, e.g., similar to a while(1) loop. By disabling RT-throttling, we rule out the possibility that latency observed during our experiments leads back to RT-throttling preempting a high-priority task (or container).
Disabling Hyperthreading Hyperthreading can cause non-deterministic latency due to shared processor resources between the hardware threads of a core. However, note that the Intel Atom CPU in the edge computing device used in our experiments does not support hyperthreading in the first place.
CPU Performance Scaling (p-states and c-states) CPU performance scaling optimizes the power consumption of the processor by adjusting its performance state (p-state). Unfortunately, the resulting frequency scaling can cause non-deterministic timing. The processor c-states, in turn, optimize power consumption during CPU idle times by changing into lower power consumption states.

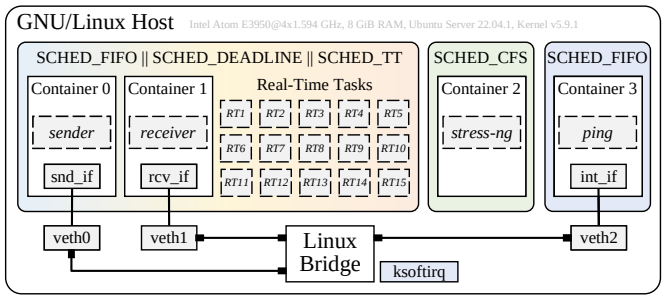

Figure 4: The basic GNU/Linux host configuration used in all experiments.

However, waking up from such idle states can again cause unpredictable latency. As a result, we disable p-states and c-states altogether.

Linux Performance Governor The use of p-states and c-states is related to the Linux performance governor that scales CPU frequency depending on the system load and idle state to balance power consumption and performance. Setting the Linux performance governor to always maximize the CPU frequency prevents varying task execution times due to CPU frequency scaling.

Raised Software Interrupt Priority Raising the priority of ksoftirq kernel threads scheduled by SCHED_FIFO above the CPU background system load ensures that the Linux kernel's network subsystem, namely, software interrupt handlers have priority over other applications scheduled by SCHED_FIFO.

Processor Core Affinity We configured the real-time tasks and the sender and receiver applications in containers 0 and 1 with a processor core affinity so the scheduler would exclusively schedule them on processor core one. The background system load and the network interference workload are balanced across all cores.

4.1.2 Linux Time-Aware Bridge Configuration. The default Linux bridge configuration used in the experiments consists of three virtual Ethernet devices [2] $veth_0$ to $veth_2$. They are connected to their respective counterparts snd_{if}, rcv_{if}, and int_{if}, which are attached to the containers. First, the snd_{if} links to $veth_0$ and connects the sender container 0 to the Linux bridge. Next, the rcv_{if} links to $veth_1$ and connects the receiver container 1 to the bridge. Finally, the int_{if} links to $veth_2$ and connects to the network interference container 3. For time-aware operations, we extend the default bridge configuration by TAPRIO qdiscs, introducing two traffic classes for scheduled Qbv traffic and best-effort (BE) traffic. For applying qdiscs to veth interfaces, there are two options depending on the logical boundaries of the Linux bridge:

Linux Bridge with Scheduled Egress Port If we consider the Linux bridge logically resembling an actual hardware bridge, the interfaces $veth_0$ to $veth_2$ act as ports of a software bridge with clear boundaries. As a result, for network traffic from containers 1 and 3 to container 2, we consider $veth_0$ and $veth_2$ as ingress interfaces and interface $veth_1$ as egress. According to the IEEE 802.1 Qbv standard, we should apply the gate control list (GCL) and the time-aware shapers to the interface facing congestion. Therefore, we apply the TAPRIO queuing discipline to interface $veth_1$ since we expect interfering network load to arrive from container 3 and exit the software bridge towards interface rcv_{if} via interface $veth_1$. We evaluate this configuration in Section 4.2.2.

Linux Bridge with Scheduled Ingress Ports The previous view on the Linux bridge considers the interfaces $veth_0$ to $veth_2$ as ports of a software bridge, assuming clear boundaries between the Linux bridge and the remainder of the kernel. However, in practice, what we perceive as a distinct software component from user space is a composition of several kernel mechanisms and user space tools that enable MAC- or IP-based forwarding of network packets between software network interfaces provided by the kernel to user space. As a result, the boundaries between the Linux bridge and the remainder of the kernel turn blurry. Thus, strictly restricting the application of queuing disciplines to interfaces identified as egress ports becomes dispensable since their identification as egress or ingress ports is ambiguous due to the Linux bridge's compositional character. Therefore, we apply the TAPRIO qdiscs on the two interfaces $veth_0$ and $veth_2$ forwarding data to the receiving container 1 via interface $veth_1$. We evaluate this configuration in Section 4.2.2.

4.1.3 Workloads. We used four containerized workloads and 15 real-time tasks for our experiments. The sender and receiver workloads in containers 0 and 1 communicate with each other utilizing the Linux bridge. Container 3 in turn, executes a ping flood trying to interfere with the sender's and receiver's communication. Lastly, container 2 executes stress –ng with 4 CPU workers to generate background system load. In addition to the four containerized workloads, we deployed 15 real-time tasks that execute with the same or higher priority as the sender and receiver applications demonstrating the effect of competing co-located high-priority tasks on the end-to-end latency between containers 0 and 1. Subsequently, we describe the functionality and rationale behind the particular workloads and task configurations in more detail.

Real-Time Tasks We have implemented real-time tasks following the guidelines on periodic real-time tasks on Linux provided in [8]. A real-time task instance periodically calculates the square root of a static sequence of numbers in a finite loop. Periodic real-time tasks take a period and a $WCET$ as parameters. We select the actual execution time at runtime by applying an exponential tail Gumbel distribution up to the WCET for the competing real-time tasks. This distribution has proven useful in the statistical analysis of task execution times [11, 55]. In total, we spawn 15 real-time tasks that utilize the processor to approximately 70% under the worst-case assumption. We always schedule the real-time tasks with the same RT scheduler (SCHED_FIFO, SCHED_DEADLINE, or SCHED_TT) as the sender and receiver applications in containers 0 and 1. Across all experiments, we used a realistic period selection for the 15 real-time tasks as described in [37] adhering to periods found in automotive applications. Additionally, for SCHED_FIFO, we configured static priorities equal to or higher than the sender and receiver, and for SCHED_TT, we generated a static schedule table incorporating preemption of the receiver.

Sender and Receiver Applications We schedule the sender and the receiver applications executing in containers 0 and 1 using the

[2] https://man7.org/linux/man-pages/man4/veth.4.html

same real-time scheduler as for the 15 real-time tasks. We enforce a sender's and receiver's period of 10 ms for SCHED_FIFO and SCHED_DEADLINE by sleeping for $(10 - WCET)$ ms. In the case of SCHED_TT, the sender and receiver periods are enforced by the scheduler. The sender application takes a timestamp t_s and executes a synthetic load identical to the real-time tasks for $WCET_{send}\mu s$ before transmitting one multicast measurement packet per hyper-period to the receiver application. The payload of the measurement message contains the previously taken user space timestamp t_s and a buffer for ten trace events consisting of a timestamp and a string that are populated by trace points in the Linux kernel. This custom trace extension enables us to timestamp the packet as it passes the Linux kernel from the sending container to the receiving container allowing detailed analysis of introduced kernel latency. On the receiver side, the receiver reads the measurement packet from interface rcv_{if} and executes the same synthetic load as the real-time tasks and the sender for $WCET_{recv}\mu s$. At the end of the receiver time slice, a timestamp t_r in user space is created, and the kernel tracing timestamps from the packet payload are extracted. Finally, the receiver application calculates the end-to-end latency $t_r - t_s$ and writes retrieved raw timestamps and latency to disk for later analysis.

System Load We use `stress-ng -class cpu -all 4` spawning 4 instances of each available CPU stressor in container 2 to generate high background load and test the temporal isolation. Therefore, we always schedule the background system load with CFS.

Network Interference The network interference workload running in container 2 generates congestion on the Linux bridge and the receiving container so that we can demonstrate the effectiveness of the time-aware Linux bridge configurations. We use a ping flood chrt −f 40 ping −f −s 6000 to send ICMP echo requests of 6000 bytes. We always schedule the network interference container with SCHED_FIFO and priority (40) higher than the background load, ensuring that it will send ICMP requests as fast as possible.

4.2 Results

We performed three experiments given the experimental setup described in the previous section to investigate the effect of the Linux bridge configuration and the kernel scheduling policy on the end-to-end latency. The three experiments cover different workloads and different Linux bridge configurations with varying scheduling policies (SCHED_FIFO, SCHED_DEADLINE, and SCHED_TT). In an initial experiment, we ran the sender and receiver connected by a default Linux bridge with a CPU load running in the background scheduled with CFS to establish an end-to-end latency baseline. In the second experiment, we added 15 RT tasks to the setup so they *compete for processor time* with the sender and receiver. In a final experiment, we varied the Linux bridge configuration (default, scheduled egress, and scheduled ingress) and introduced an additional ping flood workload to test the effect of interfering network load on the end-to-end latency.

For all experiments, in the case of SCHED_FIFO, we assigned static priority 60 to the sender and receiver and random priorities equal or higher to the 15 RT tasks. In contrast, for SCHED_DEADLINE the dynamic priorities directly result from the WCETs and periods so that no additional configurations were

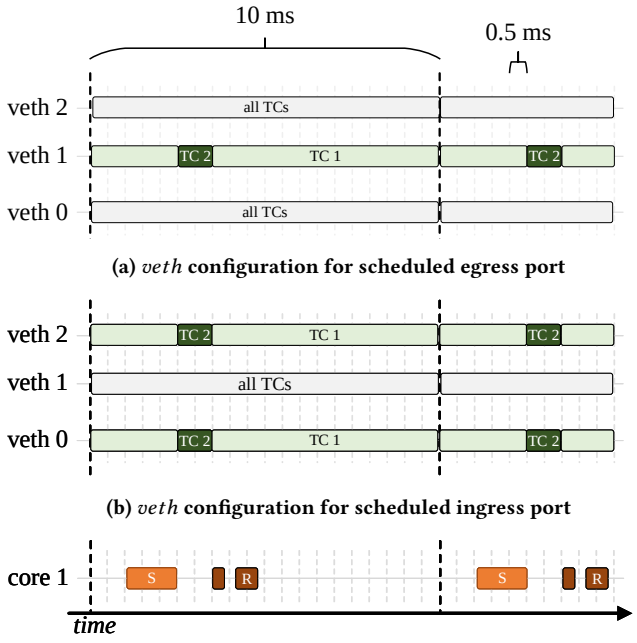

Figure 5: Showing the static schedule of sender and receiver for SCHED_TT and the schedules of the Linux bridge with scheduled egress versus scheduled ingress ports.

required. For SCHED_TT, we generated a schedule table that executes the sender consecutively at offset 1100 μs for 1400 μs while the receiver's execution starts at offset 3500 μs, is then preempted for 300 μs at offset 3900 μs and completes at 4200 μs resulting in an expected end-to-end latency of 3.1 ms. We visualize the static schedules of the sender and receiver for SCHED_TT in Fig. 5c. We ran the experiments for 10 minutes with 12 separate runs to guarantee a fair distribution of emergent runtime behaviors depending on the respective initial system state, specifically the scheduler state. Note that we are not biasing results so that SCHED_TT outperforms SCHED_FIFO and SCHED_DEADLINE by choosing a beneficial task set. However, it is an inherent property of table-driven scheduling that it enables us to optimize for certain task and communication characteristics, e.g., end-to-end latency jitter, by computing exact task dispatch and packet transmission/receive times before runtime. SCHED_FIFO and SCHED_DEADLINE can also be aligned to the transmission schedule, but only for a limited number of applications. For example, for SCHED_FIFO, the sender task can be put to the highest priority and aligned to the transmission time of the corresponding message; however, if multiple such critical communicating tasks are present, they will interfere with each other and cannot be easily aligned to the network schedule. Hence, our setup corresponds to a system where the critical tasks compete with each other both for computing resources and alignment to the communication schedule. For time-triggered scheduling, this competition for resources is resolved offline by the static schedule synthesis. In

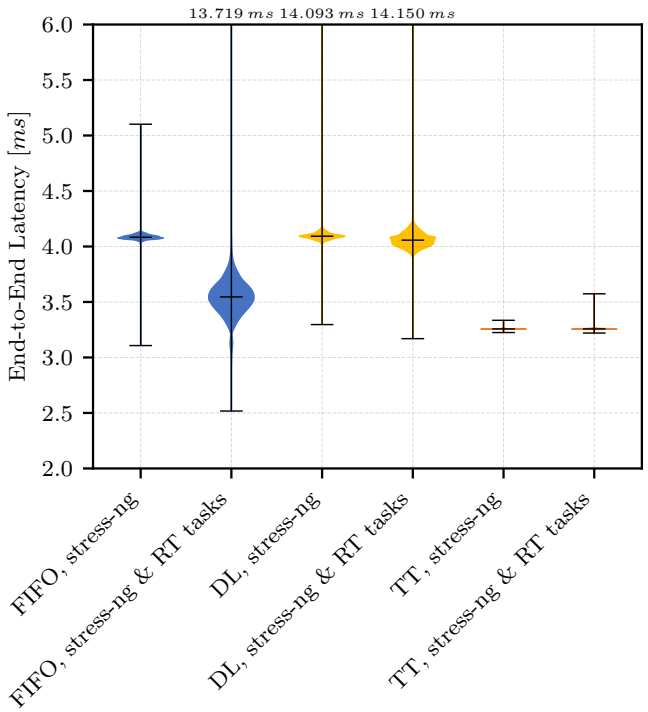

Figure 6: End-to-end latency using default Linux bridge.

terms of schedulability, it is known that EDF and TT have the highest schedulability bound (100%) while FP has a utilization bound of 69% in theory. However, in some cases, e.g., for harmonic periods, the utilization bound for FP is also 100% [47]. Here, we focus on the practical differences between the different schedulers in terms of latency and jitter within the Linux kernel.

4.2.1 Effect of Competing RT Tasks on E2E Latency. For our initial experiment, we configured the host Linux kernel to use a default Linux kernel bridge without TAPRIO qdiscs, as shown in Fig. 4. We tested a total of six configurations with the default Linux bridge varying the used Linux kernel scheduling policy (SCHED_FIFO, SCHED_DEADLINE, and SCHED_TT) and the workloads. We measured the end-to-end latency when there is only a CPU-intensive background load running CFS and when there are 15 other RT tasks present. This enabled us to identify the effect of kernel task scheduling policies on the end-to-end latency when there are competing RT tasks present, and there is no TAPRIO qdisc in place.

In Fig. 6, we visualize the measured end-to-end latency for SCHED_FIFO (FIFO), SCHED_DEADLINE (DL), and SCHED_TT (TT) with background load only and with additional RT tasks. For background load only (stress –ng), we measured a similar end-to-end latency of SCHED_FIFO $avg_{FIFO} = 4.085$, $std_{FIFO} = 0.024\,ms$, $min_{FIFO} = 3.107\,ms$, $max_{FIFO} = 5.102\,ms$ and SCHED_DEADLINE $avg_{DL} = 4.094$, $std_{DL} = 0.314\,ms$, $min_{DL} = 3.296\,ms$, $max_{DL} = 14.093\,ms$ while for SCHED_TT we measured an end-to-end latency of $avg_{TT} = 3.302$, $std_{TT} = 0.159\,ms$, $min_{TT} = 3.242\,ms$, $max_{TT} = 3.486\,ms$ that deviates at worst by $386\,\mu s$ from the offline calculated end-to-end latency. When adding competing RT tasks, we observe that the average of SCHED_FIFO dropped to $avg_{FIFO} = 3.545\,ms$ while the standard deviation increased to $std_{FIFO} = 0.161\,ms$. The drop of SCHED_FIFOs average end-to-end latency can be explained by the sender and receiver being scheduled closer to each other. This is due to the competing RT tasks delaying the sender execution when compared to the experiments with background load only. For all three schedulers, adding competing RT tasks introduces end-to-end latency outliers of exactly one sender and receiver period (10 *ms*). In the case of SCHED_DEADLINE, the outlier was even present in the experiment with only background load. We can trace back these outliers to the sender or receiver allocating a socket kernel buffer (`struct sk_buff`) for transmission or reception of a measurement packet. If the kernel's object cache is exhausted, the attempt to allocate a `struct sk_buff` causes a preemption of the RT task and a kernel worker thread `rcuc` inheriting the RT task's priority to free objects from the kernel's object cache. Once `rcuc` returns control back to the original RT task, there is not sufficient execution time left, so the RT task misses its deadline and completes its execution in the next period.

Notably, except for the single explicable and preventable outlier with $max_{TT} = 13.306\,ms$, SCHED_TT depicts no degradation of end-to-end latency when facing competing RT tasks confirming its well-known properties of temporal isolation, complex timing requirements, determinism, and predictability in an implementation for a general purpose OS such as GNU/Linux.

4.2.2 Effect of the Linux Bridge Configuration on E2E Latency. In the previous Section, we confirmed that background workloads scheduled with CFS do not degrade the end-to-end latency of RT tasks. However, competing real-time tasks with equal or higher priority affect the end-to-end latency when using SCHED_FIFO or SCHED_DEADLINE, even if we ignore the outliers present for all three schedulers. In contrast, the end-to-end latency of RT tasks scheduled using SCHED_TT is unaffected except for a single outlier.

Next, we evaluated the end-to-end latency when using different Linux bridge configurations in the presence of competing real-time tasks and interfering network load in container 2. For the first setup, we used the default Linux bridge configuration without TAPRIO qdiscs, as in the previous experiments. For the second setup, we configured the egress port of the Linux kernel bridge to apply a TAPRIO qdisc as introduced in Section 4.1.2 to prevent congestion on the receiver interface rcv_{if} in the case of container 2 sending packets to the receiver. In Fig. 5a, we illustrate the used TAPRIO qdisc on $veth_1$ and the SCHED_TT static schedule table executing on core 1 (we used the same TT schedule as in experiment 0 and visualized in Fig. 5c). Scheduled traffic is assigned to traffic class 2 while unscheduled traffic is assigned to traffic class 1. For the final setup, we configured the interface $veth_1$ and $veth_3$ attaching to the Linux kernel bridge to apply TAPRIO qdiscs to enforce packet transmission times as shown in Fig. 5b. We force all traffic leaving the interfering container 3 to be assigned to traffic class 1 so that its transmission on $veth_2$ stops during the scheduled traffic's time slot, preventing congestion on $veth_1$.

In Fig. 7, we illustrate the measured end-to-end latency for SCHED_FIFO (FIFO), SCHED_DEADLINE (DL), and SCHED_TT (TT) and the three Linux bridge configurations. Firstly, for SCHED_FIFO, we measured an end-to-end latency across Linux

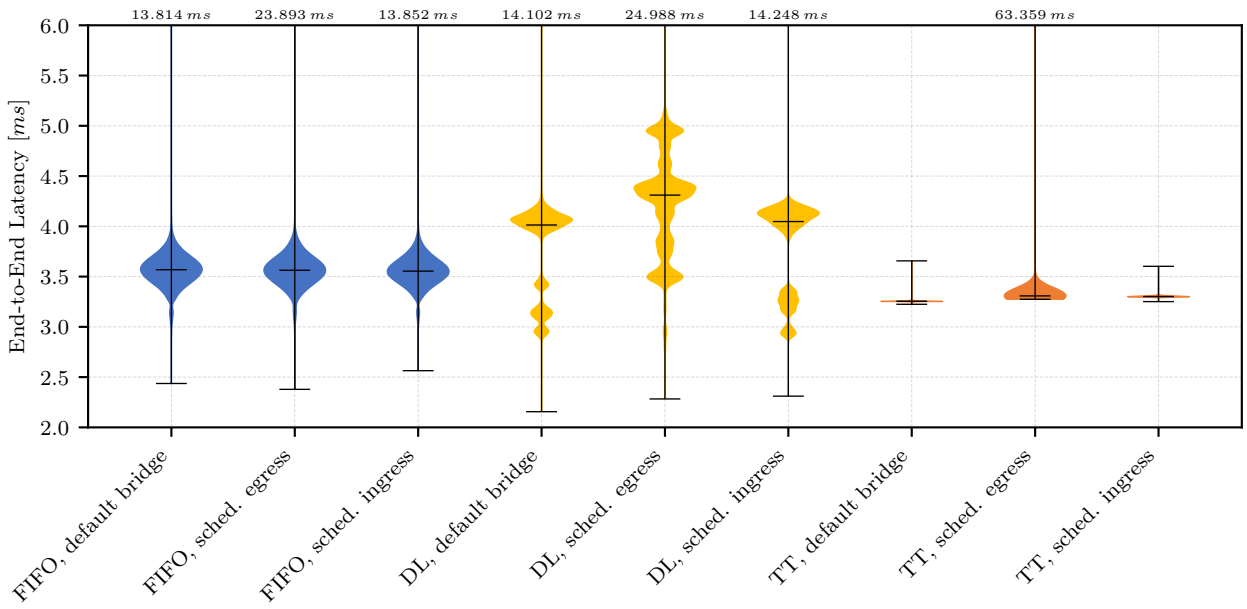

Figure 7: Comparing different bridge configurations for SCHED_FIFO, SCHED_DEADLINE, and SCHED_TT with background load, competing RT tasks, and a ping flood workload in a co-located container.

bridge configurations consistent with the results in the previous experiment featuring competing RT tasks. Note that this runtime behavior traces back to our specific task set and the rate monotonic priority assignment shifting the execution of the sender closer to the receiver resulting in the observed consistent end-to-end latency distribution. However, in the case of SCHED_DEADLINE, we see a more widespread distribution of values that we account to introducing interfering network load in the form of the ping flood workload in container 2. Note that on a containerized system introducing network load between co-located containers does materialize in increased utilization of kernel resources, namely, kernel memory and computational resources such as software interrupts with their corresponding timers and interrupt handler kernel thread. Therefore, we hypothesize that the observed end-to-end latency distribution of SCHED_DEADLINE stems from changing system behavior due to unpredictable runtime effects, such as priority inheritance and kernel memory management, that emerge increasingly under dynamic priority scheduling causing a branching out of the kernel's state space.

Finally, for SCHED_TT, we find results analogous to the previous experiment featuring competing RT tasks. Interestingly, to begin with, we observe a degradation of end-to-end latency timing determinism for SCHED_TT when using a bridge with a scheduled egress port featuring an extreme outlier of 63.359 *ms*. In contrast, for SCHED_TT with scheduled ingress ports, we measured an end-to-end latency with minimal jitter and none of the before commonly observed outliers. We repeated the experiment with SCHED_TT and scheduled ingress port to verify this result and obtained approximately the same end-to-end latency measurements.

4.2.3 Comparison to the LRT Table-Driven Scheduler. LITMUS^{RT} [13] extends the Linux kernel (the latest kernel version supported is v4.9.30) with real-time scheduling capabilities focusing on multicore scheduling and synchronization. It implements modular scheduler plugins also supporting table-driven scheduling based on time reservations similar to our SCHED_TT implementation. We compare our implementation with LITMUS^{RT} concerning the intrusiveness of the respective scheduler given by the number of kernel modifications in lines of code (LoC) and the measured end-to-end latency as in the previous experiments in Fig. 8. To compare against LITMUS^{RT}, we had to switch our experimental platform to an Intel NUC7i5BNH with an Intel Core i5 and 8 *GiB* of main memory since we were unable to boot LITMUS^{RT} v4.9.30 on our Intel Atom based edge computing device. On the Intel NUC, we ran Ubuntu server 16.04.7 with LITMUS^{RT} kernel version v4.9.30, and we applied the Linux host configurations as described in Section 4.1.1 which included disabling hyperthreading leaving us with two physical cores. Furthermore, the Linux kernel v4.9.30 lacks support for TAPRIO qdiscs, so that we could only perform experiments featuring the default bridge. We repeated the same experiments on this platform for our SCHED_TT implementation.

When comparing the intrusiveness of the LITMUS^{RT} extension with our SCHED_TT implementation, we find that LITMUS^{RT} requires modifications in 27 kernel source files totaling to $\approx 20K$ LoC with the table-driven dispatcher accounting for 426 lines of code. In addition, there are $\approx 10K$ LoC for user space libraries and tooling. In contrast, our SCHED_TT implementation requires modifications to 19 kernel source files totaling to $\approx 3.5K$ LoC with the table-driven scheduler accounting for 436 LoC. We note that LITMUS^{RT} is more

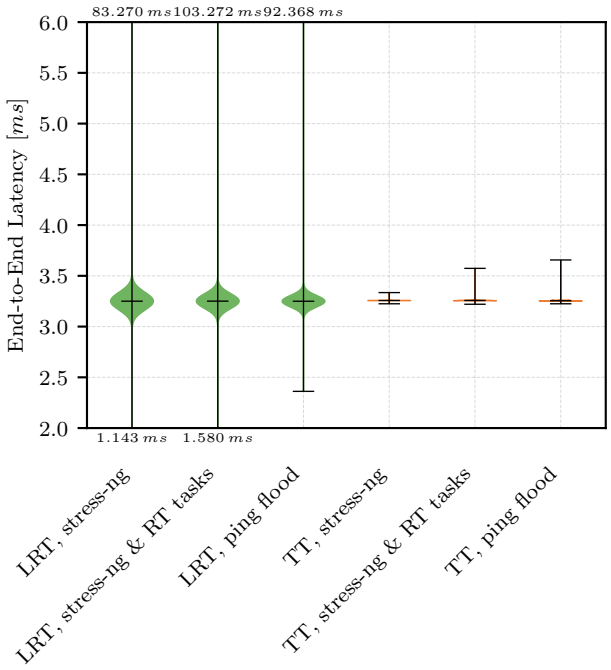

Figure 8: Comparing the end-to-end latency of LITMUS^{RT} table-driven scheduling and our SCHED_TT implementation.

intrusive to the kernel requiring more modifications, mainly because of the various multicore schedulers and experimental features provided. Our SCHED_TT implementation is, in turn, more streamlined to only time-triggered scheduling and, thus, less intrusive to the kernel facilitating porting SCHED_TT to different kernel versions (the latest supported version is v5.9.1).

In Fig. 8, we compare LITMUS^{RT}'s table-driven scheduler (LRT) with our SCHED_TT implementation (TT) replicating the exact setups from Sections 4.2.1 and 4.2.2 featuring only the default bridge. We used exactly the same task sets and schedule tables for both implementations. We note that LITMUS^{RT}'s table-driven dispatcher shows a more spread-out end-to-end latency distribution across all configurations. In contrast, SCHED_TT performs comparably to previous experiments on our edge computing device. Notably, in this set of experiments, we did not observe a single outlier of SCHED_TT. For LITMUS^{RT}, we observed frequent sender and receiver deadline misses indicating that LITMUS^{RT} would possibly require more fine-tuning of the configuration and the schedule tables. Furthermore, the performance difference might be explained by LITMUS^{RT}'s lack of PREEMPT_RT patches resulting in more and longer nonpreemptible kernel sections, which can lead to an increased end-to-end latency jitter.

4.3 Outlier discussion

In all experiments with background load (stress –ng and ping flood), we see occasional outliers for all real-time scheduling classes indicating that they are not inherent to our SCHED_TT implementation but present independently. We have investigated the source of these outliers using the Linux kernel's ftrace utility and identified two types of delays introduced by the Linux kernel that result in deadline misses.

Firstly, in the case of outliers in the range of one or two periods ($\approx 10\,ms$ to $\approx 20\,ms$), there is the issue mentioned above when an RT task attempts to allocate a *struct sk_buf* and the kernel's object cache for fast memory allocations is exhausted. Subsequently, the lower priority rcuc kernel thread inherits the RT task's priority and starts freeing the kernel object cache hence consuming the parent RT task's runtime. Once the kernel lowers rcuc's priority back and execution returns to the parent RT task, it has already exceeded its WCET causing a deadline miss and measurement packet delivery in the next period.

Lastly, we traced back the $63.359\,ms$ outlier observed in the case of SCHED_TT with scheduled egress port (c.f. Fig. 7) to software interrupt handling. In fact, the Linux kernel with PREEMPT_RT patches handles pending software interrupts in the context of any task, including background tasks with their current priority and scheduling class. If the kernel exceeds the time threshold of $\approx 2\,ms$ for in-task context software interrupt handling, the kernel delegates processing of the remaining software interrupts to the ksoftirq kernel thread. Now, if a background task scheduled with SCHED_CFS is handling software interrupts, a higher priority task becoming ready will preempt software interrupt handling and delay interrupt delivery. We argue that the absence of this type of delay in the case of scheduled ingress ports is due to the temporal proximity of transmitting the measurement packet in the container and the kernel processing the packet and corresponding software interrupts in the RT context of the sender. In contrast, in the case of scheduled egress ports, processing the software interrupt (associated with sending scheduled traffic) takes place several hundreds of microseconds later, and thus in the context of an arbitrary task, as can be seen in Fig. 5a. Furthermore, we hypothesize that the aggressive, strictly time-driven preemption of SCHED_TT promotes the occurrence of this specific delay since the receiver (or any other RT task) can preempt a lower-priority task performing in-task interrupt handling before the packet transmission interrupt has been processed.

5 CONCLUSION

In this paper, we proposed an additional kernel-level scheduler for the GNU/Linux operating system, which implements a time-triggered (TT) approach. We have investigated the benefits of our new TT scheduler in terms of the level of real-time guarantees that can be achieved. We have shown in a series of experiments that time-triggered scheduling in the Linux kernel results in reduced latency and jitter for real-time applications when compared to the existing (real-time) scheduling policies and a user-space time-triggered implementation. Additionally, in terms of end-to-end communication latencies for distributed real-time applications, we have investigated a software-based IEEE 802.1Qvb time-aware gating implementation for time-sensitive networking (TSN), where the networks and task schedules can be aligned. While our time-triggered scheduler at the kernel level has benefits in terms of stricter deterministic execution of tasks, there are still problems in the communication subsystem of Linux related to bottlenecks with certain kernel buffer allocations, resulting in occasional outliers independently of the scheduler policy used.

REFERENCES

[1] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. 2019. Container-Based Real-Time Scheduling in the Linux Kernel. *SIGBED Rev.* 16, 3 (2019), 33–38. https://doi.org/10.1145/3373400.3373405

[2] L. Abeni and G. Buttazzo. 1998. Integrating multimedia applications in hard real-time systems. In *Proc. RTSS*. https://doi.org/10.1109/REAL.1998.739726

[3] L. Abeni and G. C. Buttazzo. 2004. Resource Reservation in Dynamic Real-Time Systems. *Journal of Real-Time Systems* 27, 2 (2004), 123–167.

[4] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. 2002. A measurement-based analysis of the real-time performance of linux. In *Proc. RTAS*. 133–142. https://doi.org/10.1109/RTTAS.2002.1137388

[5] Mohammad Ashjaei, Lucia Lo Bello, Masoud Daneshtalab, Gaetano Patti, Sergio Saponara, and Saad Mubeen. 2021. Time-Sensitive Networking in automotive embedded systems: State of the art and research opportunities. *JSA* 117 (2021), 102137. https://doi.org/10.1016/j.sysarc.2021.102137

[6] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. 2016. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *Proc. RTCSA*. https://doi.org/10.1109/RTCSA.2016.41

[7] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. 2017. End-to-end Timing Analysis of Cause-effect Chains in Automotive Embedded Systems. *Journal of Systems Architecture* 80, C (2017). https://doi.org/10.1016/j.sysarc.2017.09.004

[8] Björn Brandenburg. 2012. Liu and Layland and Linux: A Blueprint for "Proper" Real-Time Tasks. https://sigbed.org/2020/09/05/liu-and-layland-and-linux-a-blueprint-for-proper-real-time-tasks/. Accessed: 18.01.2023.

[9] Björn B. Brandenburg and Mahircan Gül. 2016. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *Proc. RTSS*. https://doi.org/10.1109/RTSS.2016.019

[10] Adrien Brun, Chunhui Guo, and Shangping Ren. 2015. A Note on the EDF Preemption Behavior in "Rate Monotonic Versus EDF: Judgment Day". *IEEE Embedded Systems Letters* 7, 3 (2015), 89–91. https://doi.org/10.1109/LES.2015.2452226

[11] Alan Burns and Stewart Edgar. 2000. Predicting Computation Time for Advanced Processor Architectures. In *Proc. ECRTS*.

[12] Giorgio C. Buttazzo. 2005. Rate monotonic vs. EDF: Judgment day. *Real-Time Systems* 29, 1 (2005), 5–26.

[13] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. 2006. $LITMUS^{RT}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proc. RTSS*. https://doi.org/10.1109/RTSS.2006.27

[14] Marco Cereia, Ivan Cibrario Bertolotti, and Stefano Scanzio. 2011. Performance of a Real-Time EtherCAT Master Under Linux. *IEEE Transactions on Industrial Informatics* 7, 4 (2011), 679–687. https://doi.org/10.1109/TII.2011.2166777

[15] Felipe Cerqueira and Björn Brandenburg. 2013. A Comparison of Scheduling Latency in Linux, $PREEMPT\_RT$, and $LITMUS^{RT}$. In *Proc. OSPERT*.

[16] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency While Preserving Responsiveness. *SIGARCH Comput. Archit. News* 41, 3 (2013), 308–319. https://doi.org/10.1145/2508148.2485949

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[18] Silviu S. Craciunas, Ramon Serna Oliver, Martin Chmelík, and Wilfried Steiner. 2016. Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks. In *Proc. RTNS*. ACM. https://doi.org/10.1145/2997465.2997470

[19] Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira, and Tommaso Cucinotta. 2020. Demystifying the Real-Time Linux Scheduling Latency. In *Proc. ECRTS (LIPIcs, Vol. 165)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 9:1–9:23. https://doi.org/10.4230/LIPIcs.ECRTS.2020.9

[20] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. 2011. ChronOS Linux: A best-effort real-time multiprocessor Linux kernel. In *Proc. DAC*.

[21] Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. 2018. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092). *Dagstuhl Reports* 8, 2 (2018), 122–149.

[22] T. Fleming and A. Burns. 2015. Investigating Mixed Criticality Cyclic Executive Schedule Generation. In *Proc. WMC*.

[23] Philippe Gerum. 2004. Xenomai - Implementing a RTOS emulation framework on GNU/Linux. *White Paper, Xenomai* (2004).

[24] Carlos San Vicente Gutiérrez, Lander Usategui San Juan, Irati Zamalloa Ugarte, and Victor Mayoral Vilches. 2018. Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications. *CoRR* abs/1808.10821 (2018). arXiv:1808.10821 http://arxiv.org/abs/1808.10821

[25] D. Hart, J. Stultz, and T. Ts'o. 2008. Real-time Linux in real time. *IBM Systems Journal* 47, 2 (2008), 207–220. https://doi.org/10.1147/sj.472.0207

[26] Pierre-Emmanuel Hladik, Anne-Marie Deplanche, Sebastien Faucou, and Yvon Trinquet. 2007. Adequacy between AUTOSAR OS specification and real-time scheduling theory. In *Proc. SIES*. https://doi.org/10.1109/SIES.2007.4297339

[27] IEEE. 2014. 802.1Q-2014 - Bridges and Bridged Networks. http://www.ieee802.org/1/pages/802.1Q.html.

[28] IEEE. 2016. 802.1Qbv - Enhancements for Scheduled Traffic. http://www.ieee802.org/1/pages/802.1bv.html. Draft 3.1, Accessed: 12.01.2023.

[29] IEEE. 2016. Official Website of the 802.1 Time-Sensitive Networking Task Group. http://www.ieee802.org/1/pages/tsn.html. Accessed: 23.10.2020.

[30] IEEE. 2017. 802.1AS-Rev - Timing and Synchronization for Time-Sensitive Applications. http://www.ieee802.org/1/pages/802.1AS-rev.html. Accessed: 12.01.2023.

[31] Infineon. 2019. Infineon and TTTech Auto collaborate to enable level 4 and level 5 automated driving. https://www.infineon.com/cms/en/about-infineon/press/market-news/2019/INFATV201901-027.html. Accessed: 12.01.2023.

[32] Intel. 2019. Configuring TSN Qdiscs. https://tsn.readthedocs.io/qdiscs.html.

[33] Damir Isović and Gerhard Fohler. 2000. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proc. RTSS*. https://doi.org/10.1109/REAL.2000.896010

[34] Damir Isović and Gerhard Fohler. 2009. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Syst.* 43, 3 (2009). https://doi.org/10.1007/s11241-009-9088-3

[35] Zhe Jiang, Shuai Zhao, Pan Dong, Dawei Yang, Ran Wei, Nan Guan, and Neil Audsley. 2020. Re-Thinking Mixed-Criticality Architecture for Automotive Industry. In *Proc. ICCD*. 510–517. https://doi.org/10.1109/ICCD50377.2020.00092

[36] Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. 2019. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *Proc. RTAS*. https://doi.org/10.1109/RTAS.2019.00009

[37] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. 2015. Real world automotive benchmarks for free. In *Proc. WATERS*.

[38] Gagan Nandha Kumar, Kostas Katsalis, and Panagiotis Papadimitriou. 2020. Coupling Source Routing with Time-Sensitive Networking. In *Proc. IFIP Networking Conference*.

[39] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. 2016. Deadline Scheduling in the Linux Kernel. *Softw. Pract. Exper.* 46, 6 (2016), 821–839.

[40] H. Lonn and J. Axelsson. 1999. A comparison of fixed-priority and static cyclic scheduling for distributed automotive control applications. In *Proc. ECRTS*. https://doi.org/10.1109/EMRTS.1999.777460

[41] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. 2012. Modular scheduling of distributed heterogeneous time-triggered automotive systems. In *Proc ASPDAC*.

[42] Alessandro Macchelli and Claudio Melchiorri. 2002. A real-time control system for industrial robots and control applicationss based on real-tiem Linux. *IFAC Proceedings Volumes* 35, 1 (2002), 55–60.

[43] Michael M. Madden. 2019. Challenges Using Linux as a Real-Time Operating System. In *AIAA Scitech 2019 Forum*. AIAA. https://doi.org/10.2514/6.2019-0502 arXiv:https://arc.aiaa.org/doi/pdf/10.2514/6.2019-0502

[44] Shane D. McLean, Emil A. Juul Hansen, Paul Pop, and Silviu S. Craciunas. 2022. Configuring ADAS Platforms for Automotive Applications Using Metaheuristics. *Frontiers in Robotics and AI* 8 (2022), 353. https://doi.org/10.3389/frobt.2021.762227

[45] Ayhan Mehmed, Wilfried Steiner, and Maximilian Rosenblattl. 2017. A Time-Triggered Middleware for Safety-Critical Automotive Applications. In *Ada-Europe*.

[46] Anna Minaeva and Zdeněk Hanzálek. 2021. Survey on Periodic Scheduling for Time-Triggered Hard Real-Time Systems. *ACM Comput. Surv.* 54, 1 (2021). https://doi.org/10.1145/3431232

[47] Morteza Mohaqeqi, Mitra Nasri, Yang Xu, Anton Cervin, and Karl-Erik Årzén. 2018. Optimal harmonic period assignment: complexity results and approximation algorithms. *Real Time Syst.* 54, 4 (2018), 830–860.

[48] Georg Niedrist. 2018. Deterministic architecture and middleware for domain control units and simplified integration process applied to ADAS. In *Fahrerassistenzsysteme 2016*. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-21444-9

[49] A.K. Parekh and R.G. Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking* 1, 3 (1993), 344–357. https://doi.org/10.1109/90.234856

[50] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. 2019. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Comput. Surv.* 52, 1 (2019). https://doi.org/10.1145/3297714

[51] Florian Sagstetter, Sidharta Andalam, Peter Waszecki, Martin Lukasiewycz, Hauke Stähle, Samarjit Chakraborty, and Alois Knoll. 2014. Schedule Integration Framework for Time-Triggered Automotive Architectures. In *Proc. DAC*. ACM.

[52] Claudio Scordino and Giuseppe Lipari. 2006. Linux and real-time: Current approaches and future opportunities. In *Proc. ANIPLA*. Available at http://retis.sssup.it/~lipari/papers/ANIPLA_scordino_lipari.pdf.

[53] R. Serna Oliver, S. S. Craciunas, and W. Steiner. 2018. IEEE 802.1Qbv Gate Control List Synthesis using Array Theory Encoding. In *Proc. RTAS*.

[54] Lui Sha, Tarek Abdelzaher, Karl Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok.

2004. Real time scheduling theory: A historical perspective. *Real-Time Systems* 28 (2004), 101–155. https://doi.org/10.1023/B:TIME.0000045315.61234.1e

[55] Karila Palma Silva, Luis Fernando Arcaro, and Romulo Silva De Oliveira. 2017. On Using GEV or Gumbel Models When Applying EVT for Probabilistic WCET Estimation. In *Proc. RTSS*. https://doi.org/10.1109/RTSS.2017.00028

[56] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. 2015. Time-Triggered Mixed-Critical Scheduler on Single and Multi-processor Platforms. In *Proc. HPCC*.

[57] Marija Sokcevic. 2020. Partitioned Complexity. https://www.tttech-auto.com/knowledge-platform/partitioned-complexity. Accessed: 13.01.2023.

[58] TTTech Computertechnik AG. 2018. Automated Driving Offering. https://www.tttech-auto.com/motionwise. Accessed: 12.01.2023.

[59] Marek Vlk, Kateřina Brejchová, Zdeněk Hanzálek, and Siyu Tang. 2022. Large-scale periodic scheduling in time-sensitive networks. *Computers & Operations Research* 137 (2022), 105512. https://doi.org/10.1016/j.cor.2021.105512

[60] N. Vun, H.F. Hor, and J.W. Chao. 2008. Real-Time Enhancements for Embedded Linux. In *Proc. ICPADS*. https://doi.org/10.1109/ICPADS.2008.108

[61] Jia Xu. 2002. Satisfying complex timing constraints with pre-run-time scheduling. *IFAC Proceedings Volumes* 35, 1 (2002), 273–278. https://doi.org/10.3182/20020721-6-ES-1901.00951

[62] J. Xu and D.L. Parnas. 1998. Priority Scheduling Versus Pre-Run-Time Scheduling. *IFAC Proceedings Volumes* 31, 14 (1998), 53–60. https://doi.org/10.1016/S1474-6670(17)44872-5