Supporting Low-Latency, Low-Criticality Tasks in a Certified Mixed-Criticality OS

Manohar Vanga
MPI-SWS
mvanga@mpi-sws.org

Andrea Bastoni
SYSGO AG
aba@sysgo.com

Henrik Theiling
SYSGO AG
hth@sysgo.com

Björn B. Brandenburg
MPI-SWS
bbb@mpi-sws.org

ABSTRACT

This paper presents a case study in the design and implementation of OS-level support for low-criticality tasks with stringent latency requirements, a particularly challenging aspect of mixed-criticality workloads, in PikeOS, a commercial, certified mixed-criticality OS. Special consideration is placed on key real-world design constraints that arise in a commercial setting, pertaining to customer needs, vendor constraints, and certification demands.

CCS CONCEPTS

• **Computer systems organization → Real-time OS**;

KEYWORDS

mixed-criticality systems, time partitioning, resource reservations

ACM Reference format:
Manohar Vanga, Andrea Bastoni, Henrik Theiling, and Björn B. Brandenburg. 2017. Supporting Low-Latency, Low-Criticality Tasks in a Certified Mixed-Criticality OS. In *Proceedings of RTNS '17, Grenoble, France, October 4–6, 2017,* 10 pages.
https://doi.org/10.1145/3139258.3139274

1 INTRODUCTION

Mixed-criticality systems (MCS) naturally emerge in safety-critical application domains when non-functional requirements related to size, weight, and power (*SWaP*) force the integration of components with different failure-assurance levels, or *criticalities*, onto a single hardware platform [5]. The inherent challenge in such systems lies in ensuring isolation between tasks at different criticality levels that share hardware resources (*e.g.*, processor cores, memory, caches), while still ensuring that performance requirements (both throughput and latency) are met.

While a large number of approaches for analyzing mixed-criticality systems have been proposed in the real-time literature (see Sec. 6), they are predominantly based on models that assume low-criticality tasks to also be *low-importance* tasks. For example, under Vestal's model [44], one of the first and most-studied mixed-criticality models, lower-criticality tasks are no longer serviced, or serviced only in a best-effort manner, when the system enters a degraded mode.[1]

However, in commercial mixed-criticality systems, the criticality of a task simply refers to the level of assurance that it has been certified for, and does not always correspond with its importance [13]. That is, a task may be deemed to be of low criticality during safety certification, but still be important from a business or product standpoint. For example, the responsiveness of a low-criticality network driver is not necessarily less important than a higher-criticality control task if increased network latency degrades premium comfort or entertainment features (further examples are provided in Sec. 2.1). Thus, in a commercial mixed-criticality OS, in addition to ensuring OS-level timeliness guarantees for high-criticality tasks, it is equally important to provide such guarantees for low-criticality tasks, as they may still be of high importance to customers, even if they are less relevant from a safety certification point of view.

In this paper, we explore the problem of supporting mixed-criticality applications from the perspective of the vendor of an established and certified multicore real-time OS. From this point of view, the problem is not how to support high-criticality workloads, as such systems have been successfully supported for many years and deployed in the products of numerous customers across a range of safety-critical domains. Rather, the challenge lies in providing first-class OS support for low-criticality applications with demanding performance requirements, *without compromising the isolation of high-criticality tasks or system certifiability.*

In our experience, there is little customer interest in abandoning established system-design practices for the integration of high-criticality components. Consequently, there exists no strong business incentive to radically re-design the OS and its process scheduler. Rather, our goal is to integrate support for mixed-criticality workloads with as minimal a disruption to existing use-cases and workflows as possible, in particular, without affecting the system's ability to host exclusively high-criticality workloads.

This paper. We report on a case study in the design and implementation of such a system in PikeOS, a certified microkernel in

[1]As suggested by Alan Burns in his keynote at the 2015 Dagstuhl Seminar on "Mixed Criticality on Multicore/Manycore Platforms" (Seminar 15121), we use the terms "normal mode" and "degraded mode" to refer to what is in the literature often called low- and high-criticality mode, respectively, to better reflect system performance from an application point of view.

real-world use in many safety-critical domains, including avionics, automotive, and transportation applications. In Sec. 2, we provide an overview of PikeOS and identify deficiencies in its support for low-criticality tasks with stringent latency requirements (*i.e.*, tasks that can tolerate at most a few milliseconds of latency or less). We then highlight key design constraints that arise in the context of a commercial real-time kernel that are typically not considered in primarily research-oriented designs (Sec. 3). Based on this requirements analysis, we present a minimally invasive extension of the PikeOS scheduling architecture that provides explicit OS-level support for low-criticality, low-latency tasks and sketch its implementation in PikeOS (Sec. 4). Finally, we present an evaluation of the proposed design — in an open, freely shareable re-implementation of the PikeOS scheduling architecture in LITMUS^{RT}, a Linux-based research RTOS — that empirically substantiates the identified problem and shows the solution to be effective and practical (Sec. 5).

2 THE LOW-LATENCY CHALLENGE

We begin by motivating why low-criticality tasks with stringent latency requirements are an important workload, then provide an overview of PikeOS, and finally discuss why, in a mixed-criticality context, the former are challenging to support in the latter.

2.1 Motivating Use-Cases

With embedded multicore platforms gaining adoption in real-time systems, SWaP and cost considerations are driving the consolidation of an increasing variety of workloads onto shared hardware platforms. It is this consolidation that presents a problem when part of the workload involves low-criticality tasks with nonetheless demanding requirements. We give three examples of such workloads from the automotive domain that illustrate how such requirements arise, and why it is important to provide OS-level support for them.

Rear-view cameras. Consider a car with an external rear camera that provides a rear-view video feed to a screen inside the car, a nowadays common feature, where the rear-view camera is linked to the display via UDP packets that are transmitted over an automotive Ethernet connection. As component costs must be low (due to market pressures), the employed network adapters have only limited on-device buffer memory for packets. To achieve an acceptable frame rate without any dropped frames or other visual glitches, incoming packets must be processed within a few milliseconds of their arrival (*i.e.*, the network driver is latency-sensitive).

As a rear-view camera is a purely assistive technology that augments the normal rear-view mirror, and since the driver of the vehicle both remains in full control and retains full responsibility for the safe operation of the vehicle, the rear-view camera and its supporting infrastructure are not critical to vehicle safety (*i.e.*, they are "nice to have," but not essential). However, for the manufacturer of the vehicle, the rear-view camera is a crucial feature, as the error-free operation of all driver assistance features is essential to a car model's critical acclaim and thus commercial success. The rear-view camera system and the corresponding display is thus a *low-criticality* application that is of *high importance* to the application developer (*i.e.*, the RTOS customer).

Wireless audio streaming. Current-generation cars allow cell phones and other mobile devices to stream audio to the car's speakers, and to receive audio input from integrated microphones, via common wireless protocols such as Bluetooth or IEEE 802.11. Such audio features naturally require the wireless network stack and audio driver to exhibit *at most a few milliseconds of latency*, as any gaps in playback or noticeable buffering would degrade the user experience. While such functionality is certainly *not safety-critical*, it is of *high importance* from a business standpoint since users expect smooth and error-free media streaming and hands-free calls.

Touch input. Recent luxury cars incorporate touch-based infotainment systems that give passengers a central point of control over comfort and convenience features of the car. Input events generated by the touch screen are processed by a thread of the graphical user interface (GUI) framework and routed to individual applications, which then react to the input and provide some sort of visual feedback to users. To give applications sufficient time to respond, the GUI infrastructure must exhibit as little latency as possible. Again, this is an example of an application that is of *low criticality* from a safety standpoint (it is used either when the car is at rest, or only by passengers other than the driver when moving), but of *high importance* to the car maker since GUI lag (*i.e.*, observable latency from touch to visual feedback) is frustrating to users and thus highly undesirable in a premium product.

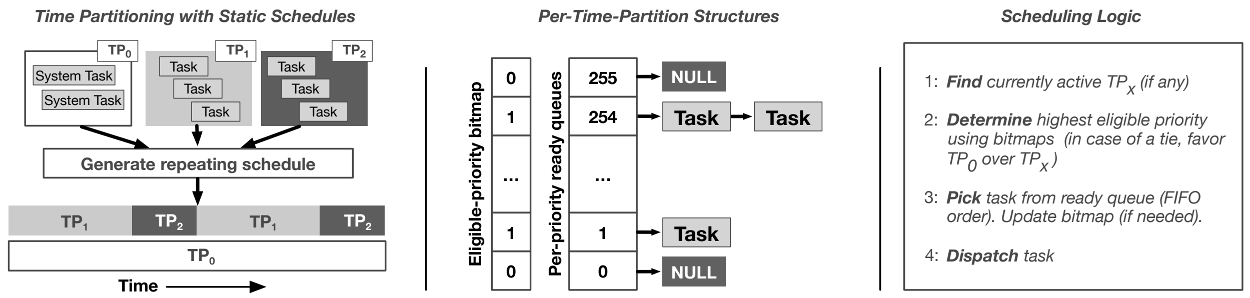
To reiterate, due to cost and SWaP pressures, there is a growing need for non-critical, but latency-sensitive workloads such as those sketched above to be co-located with more critical applications. This poses considerable challenges for the OS, as discussed next.

2.2 PikeOS and its Scheduling Architecture

PikeOS is a separation microkernel for multi-core, hard-real-time systems, and can be used as both a real-time OS (*i.e.*, hosting native applications) as well as a type-1 hypervisor (*i.e.*, hosting complete operating systems). PikeOS provides various *personalities*, or different OS interfaces (*e.g.*, ARINC 653, Linux, POSIX, AUTOSAR, *etc.*), for the development of applications in different domains.

PikeOS is widely used in industry due to its certifiable nature: PikeOS has been certified to safety standards such as DO-178B (avionics), IEC 61508 (electrical/electronic/programmable electronic safety-related systems), and EN 50128 (railways), making it a "battle-hardened," top-tier choice for mixed-criticality applications with components at different safety and security levels that need to be isolated via resource partitioning.

Scheduling in PikeOS. The PikeOS scheduler is based on time partitioning (as defined by the APEX specification in the ARINC 653 standard reference). Conceptually, time partitions are encapsulating containers for a set of threads, where threads in different time partitions are scheduled in mutually exclusive time windows. In PikeOS, application tasks are assigned to *application time partitions*, which are activated periodically as specified by a repeating static schedule. When a time partition is activated, any tasks of the previously active time partition are forcibly preempted; frequent time-partition switches thus incur significant runtime overheads. As application time partitions are strictly separated from another, tasks in different partitions can be certified to different levels of assurance (*i.e.*, each in accordance to its own criticality).



1: ***Find*** *currently active TP_X (if any)*
2: ***Determine*** *highest eligible priority using bitmaps (in case of a tie, favor TP_0 over TP_X)*
3: ***Pick*** *task from ready queue (FIFO order). Update bitmap (if needed).*
4: ***Dispatch*** *task*

Figure 1: An overview of the PikeOS scheduler. The system comprises multiple application time partitions (*TP_X*), for which a static schedule is generated, and the system time partition (*TP_0*), which is always eligible (left). Each time partition consists of 256 ready queues and an associated bitmap tracking which levels have eligible tasks (middle). At runtime, the first task from the highest-priority, non-empty ready queue, in either *TP_0* or the currently active *TP_X*, is dispatched (right).

Owing to PikeOS' microkernel design, all OS functionality (*e.g.*, device drivers, file systems, *etc.*) is implemented as service daemons (or tasks) scheduled in time partitions similarly to standard applications. To ensure that essential system services are always available, PikeOS assigns these tasks to a special *time partition zero* that is always eligible to run. Thus, in PikeOS, there may be up to two active time partitions in the system at any given time: time partition zero (TP_0) and, depending on the static schedule, an application time partition (TP_X). Note that threads in TP_0 are always certified to the highest assurance level since their functional and non-functional correctness is essential to the correct operation of the entire system.

For every time partition in the system, PikeOS maintains a ready queue for each of 256 supported priority levels, and each ready queue is simply a FIFO-ordered list of tasks eligible for scheduling at that priority level. When making a scheduling decision, the scheduler finds the highest-priority, non-empty ready queue, and picks the first task from it. It does this taking into account tasks from both TP_0 and the currently active TP_X, if any, with TP_0 tasks taking precedence over those in the currently active TP_X at any given priority level. Consequently, while the worst-case latency incurred by tasks in TP_X depends on the static schedule, TP_0 tasks are always schedulable and thus incur lower latency (affected only by other higher-priority tasks in the system).

For each time partition in the system, the PikeOS scheduler maintains a *priority bitmap* to track non-empty priority levels so that it can efficiently determine the highest priority that is currently eligible when making a scheduling decision. That is, for each time partition and priority level, a bit is set if and only if runnable threads exist in the corresponding time partition at the given priority level.

Fig. 1 summarizes the PikeOS scheduling architecture. On a multiprocessor, it is instantiated on each core (partitioned scheduling).

2.3 Mixed-Criticality Support in PikeOS

We now discuss how the PikeOS scheduler, as described above, provides support for tasks with varying criticalities and performance requirements. For simplicity, we consider four broad classes of tasks[2] (Fig. 2): tasks with either *high* or *low* criticality, combined with either *high* or *low* latency tolerance.

[2]In practice, more complex taxonomies with multiple criticality levels exist; however, two levels of criticality suffice to illustrate the low-latency, low-criticality problem.

	Low Latency Required	High Latency Acceptable
High Criticality	***TP_0***	***TP_0*** or ***high-criticality TP_X***
Low Criticality	**This paper**	***Low-criticality TP_X***

Figure 2: Mixed-criticality support in PikeOS.

High-criticality support. For high criticality tasks (*i.e.*, those that have been certified, or are expected to be certified, to the highest assurance level), two cases exist. If they do not require low latency (*e.g.*, computation-heavy, mission-critical planning tasks), one can perform a WCET analysis or estimation and place the task within an appropriately dimensioned TP_X. On the other hand, high-criticality tasks with low-latency requirements (*e.g.*, safety-critical event handlers or threads in charge of retrieving sensor data at a high rate with minimal delay) can simply be placed within TP_0 at an appropriate priority level to eliminate the added delay of having to wait for a TP_X slot activation in the static schedule.

Importantly, since TP_0 is certified at the highest criticality level anyway, this means that in general, the choice of placing a particular high-criticality task in TP_0 versus within a TP_X is a balancing act that does *not* substantially alter the overall certification burden: one must consider the tradeoff between an acceptable latency bound and system performance, but regardless of where such a task is placed, TP_0 requires certification at the highest level as it is essential to the correct operation of the entire system, and it potentially interferes with the rest of the system, thus requiring additional analysis consideration in all others TP_X's. Concerning the choice of placement, as the latency requirement of a high-criticality task becomes more demanding, there is less benefit in placing it within a TP_X, since allocating a TP_X with a short period results in a static schedule with a large number of preemptions, which consequently degrades task throughput and increases system overheads.

Low-criticality support. Similarly, for low-criticality tasks that do not require low latency (*e.g.*, navigation or route planning tasks in the bottom-right quadrant in Fig. 2), one can simply dimension a TP_X with the appropriate duration and frequency so as to satisfy the throughput and latency needs of the task.

Finally, how does one deal with the lower, left quadrant of Fig. 2: *low-latency, low-criticality tasks* (L3C tasks)? While L3C tasks can *conceptually* be placed within a TP_X dimensioned with the appropriate duration and frequency, *in practice*, the resulting high frequency of slots in the static schedule causes an unacceptable degradation of system performance and task throughput. (We demonstrate this experimentally in Sec. 5.) In addition, even if an L3C task is placed in a TP_X with an extremely high activation frequency, it still incurs higher *average* latency compared to an equivalent task in TP_0.

Alternatively, L3C tasks may be placed within TP_0. However in order to not interfere with any high-criticality tasks in TP_0 or any TP_X (*i.e.*, to guarantee *freedom-from-interference*), *criticality-monotonic priorities* [3] must be used in this case, so that L3C tasks have lower priority than any high-criticality task. Unfortunately, this again results in excessive latency due to the interference caused by the higher-priority tasks in the system. For demanding L3C tasks with stringent maximum and/or average latency requirements, this is not a viable solution. Further, giving such L3C tasks a higher priority is not an option: it would cause a massive increase in the certification burden, as they now must be shown to not cause undue interference to lower-priority, *higher*-criticality tasks *at the level of assurance of the interfered-with higher-criticality tasks.* While conceptually possible, this is not an economically viable solution.

Thus, to properly support L3C tasks in TP_0, we require a mechanism that allows a task's priority to be chosen independently of its criticality (*i.e.*, the ability to place a low-criticality task at a high priority), albeit without substantially changing the existing time-partitioning architecture, which is mandated by the ARINC 653 standard, entrenched in legacy systems, and desired by customers. To this end, this paper describes an extension of the PikeOS scheduling architecture for supporting L3C tasks without compromising isolation or certifiability of the system.

3 DESIGN CONSTRAINTS

Extending the scheduler in a real-world certified kernel like PikeOS presents certain design constraints that are usually not given much attention in purely academic settings. These constraints arise due to the need to *(i)* support previously-certified legacy applications, *(ii)* fit in with the existing ecosystem of established tools, software architectures, and system-design practices, and *(iii)* provide application developers with simple, intuitively understood mechanisms.

Taking these issues into account is crucial, as both certification of legacy applications and the incorporation of tools and workflows incur significant financial cost and up-front investment (*i.e.*, employee time spent on documentation for certification, training of application developers in the use of specialized tools, the wealth of experience gained from fielding and refining robust software architectures in prior product generations, *etc.*). While an in-depth discussion of the software engineering and business aspects of MCS support is beyond the scope of this paper, to provide an idea of the real-world context, we outline some of the primary constraints that guided the design of our solution for supporting L3C tasks.

A. Strictly opt-in for OS customers. Owing to the financial investment of the customer base of a certified kernel, the first and foremost design constraint is that any support for L3C tasks must be strictly opt-in. That is, the majority of customers that do not (yet) need L3C support should require no changes in their designs, implementations, configurations, and deployment workflows.

B. Minimally intrusive for the OS vendor. Certification of commercial real-time kernels requires a large investment of both time and money. Consequently, any extensions to the kernel, as is needed to provide support for L3C tasks, must aim to minimize re-certification efforts for vendors, which means that fundamentally changing the core scheduling architecture is not an option.

C. Runtime efficiency. Increases in runtime overheads must be avoided, at least for customers that do not opt in to the L3C support features. For customers opting in to the new features, the runtime costs should still be as low as possible (*e.g.*, requiring new hardware to support the added overhead is undesirable, while requiring adjustments to the processor allocations of time partitions to account for new overheads or bounded interference is considered tolerable).

D. Support for static configuration. Certified safety-critical systems are commonly statically configured. For example, PikeOS provides an Eclipse-based IDE that allows for static configuration of all relevant parameters for the entire system. In general, a special developer role exists, the *system integrator*, whose responsibility it is to determine the partition schedule and all other resource partitions (memory, communication end points, *etc.*), and to integrate components developed by independent teams into a complete system. To fit into existing practices and established engineering processes, our extensions should continue to support this separation of roles and the ability to statically configure the system.

E. Analytically sound. Our design should be amenable to analysis. In particular, one should be able to bound *a priori* how much additional interference is introduced by L3C tasks on both higher- and lower-criticality tasks in any time partition in the system.

F. Established approaches. To ease the certification burden, it is desirable for the design to rely on established and mature techniques that are well-understood, widely accepted, and that have been proven to be practical and reliable in prior systems and prototypes.

G. Low barrier to entry. To ensure acceptance by the customer base, any added feature should be easy to opt in to (*i.e.*, without requiring system-wide changes), easy to use once it has been adopted, and transparent to latency-insensitive tasks and time partitions. In particular, the added support for L3C tasks should provide strong isolation guarantees without requiring the system integrator or individual developers to fully understand all relevant scheduling theory, and without requiring any manual adjustments to time partitions that do not require the new features.

H. Optionally strict freedom-from-interference. Depending on the specific certification process, it often sufficient to ensure freedom from *unbounded* interference. That is, typically, *some* interference is acceptable as long as it can be shown *a priori* to be bounded, at a level of assurance commensurate with the criticality of the interfered-with task. However, certain maximum-importance tasks (either in TP_0 or a TP_X) must be strictly isolated from L3C tasks. That is, it is insufficient to guarantee merely *bounded* interference for such tasks; rather, it must still be possible to ensure *zero* interference for them without forgoing all benefits of L3C support.

I. Support for customer-specific specialization. Finally, due to the extremely varied and hard-to-anticipate nature of project-specific customer requirements, there is a need to support customer-defined specialization and customization in L3C support. The challenge here is that per-customer specialization must be economically viable, which means that it should not invalidate the existing

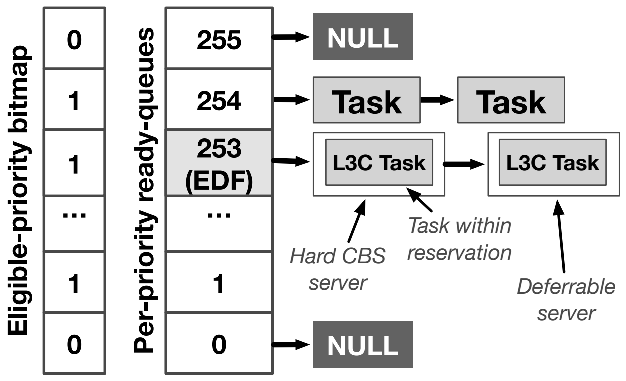


Figure 3: The proposed solution, which satisfies the constraints discussed in Sec. 3 by encapsulating L3C tasks within reservations in EDF-designated priority levels.

documentation packages for most parts of the kernel that are unconcerned with scheduling and L3C support.

4 DESIGN AND IMPLEMENTATION

In accordance with the discussed design constraints and business incentives, we did not seek to invent a radically new scheduling approach. To the contrary, we iterated on a *simple* design that leverages as much as possible *existing* state-of-the-art techniques to satisfy mixed-criticality workloads in a production RTOS.

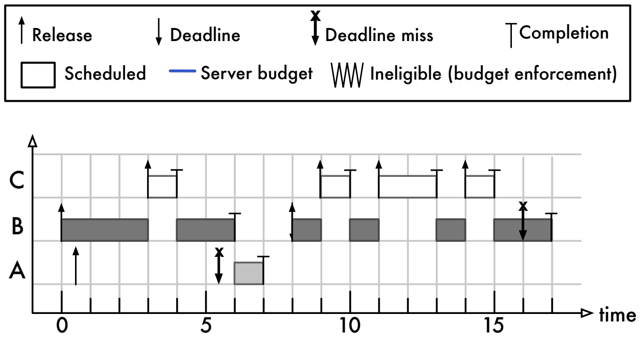
At a conceptual level, our approach is based on encapsulating L3C tasks within hard *resource reservations* [29] that are scheduled via an *earliest-deadline-first* (EDF) scheduler[3] using an "EDF within fixed priorities" approach [20], which allows L3C tasks to be safely placed in TP_0 at any priority level without fundamentally increasing the overall certification burden. At the implementation level, our solution comprises three key components: *(i)* a scheduler plugin framework with a well-defined interface that is amenable to certification, *(ii)* a reservation-based EDF plugin that hooks into the framework, and *(iii)* a flexible API within the reservation plugin that allows implementing customer-specific reservation policies.

In the following, we discuss this architecture in detail, provide an illustrative example highlighting key properties, and assess how the design satisfies the constraints in Sec. 3. Additionally, we provide a high-level sketch of the prototype implementation in PikeOS.

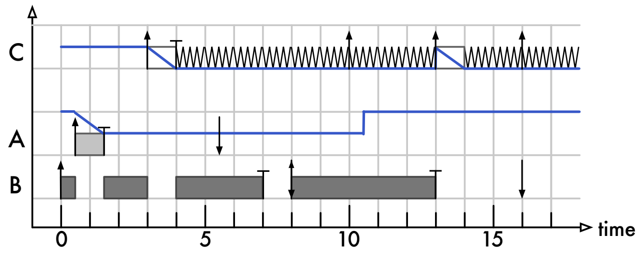
4.1 EDF-Scheduled Reservations in FP Bands

Our overall solution, as illustrated in Fig. 3, involves introducing reservations [29] (also known as *servers*), *i.e.*, analytically sound containers for threads, at specific priority levels, called *EDF bands*, within TP_0 that are scheduled via EDF. The ready queue corresponding to an EDF band is ordered by non-decreasing server deadlines. Whenever an EDF band is the highest eligible priority, a task from the first reservation (*i.e.*, the reservation with the earliest deadline) is dispatched. Similar "EDF in priority bands" approaches have been studied extensively in prior work [20]. A key advantage in our

[3]While a similar approach for fixed-priority systems, called *adaptive mixed-criticality* (AMC) scheduling [3] has been proposed in prior work, we focus on an EDF-based solution as it enables greater flexibility, higher schedulable utilization of the system [28], and since it does not require dedicating a large number of priority bands to L3C tasks.



(a) Example scenario under the default PikeOS scheduler without L3C support.



(b) The same scenario under PikeOS with the proposed L3C support.

Figure 4: Example of the benefits of the proposed design: low latency for L3C tasks without unbounded interference.

context is that EDF bands require minimal changes to the scheduling architecture: from the perspective of the PikeOS fixed-priority scheduler, we are simply deferring the queuing logic for certain priority levels to a secondary EDF scheduler, instead of the default FIFO scheme. This is in accordance with design constraint B.

Since no single reservation policy is ideal for all workloads (design constraint I), we chose to avoid hard-coding a policy. Rather, within the reservation scheduler, we define a sub-API that allows well-known reservation schemes such as polling, sporadic [38, 40, 41], deferrable [42], or hard constant-bandwidth servers [1], as well as custom reservation types, to be implemented.

Note that reservations are supported *only* in TP_0 and *not* in any TP_X — our design is intended specifically for L3C tasks with latency requirements that cannot be accommodated by the cyclic TP_X schedule, the primary temporal enforcement mechanism. Since we are not aware of a pressing use-case for reservations within a TP_X, we omitted this capability to minimize the required changes.

Example. Before elaborating how the design satisfies the considerations in Sec. 3, we illustrate the approach with the example shown in Fig. 4. Consider a simplified scenario with three tasks with *(i)* a low-priority L3C task in TP_0 (task A in Fig. 4) with a WCET of 1 ms and a relative deadline of 5 ms, *(ii)* a medium-priority, high-criticality task with a WCET and period of 5 ms and 8 ms, respectively, which resides in a TP_X with 100% utilization (task B), and *(iii)* a high-priority L3C task in TP_0 with a WCET of 1 ms and a deadline of 10 ms (task C). For the sake of the example, suppose that task C occasionally suffers transient overloads with bursty arrivals.

First consider the task set under the stock PikeOS scheduler as deployed today without L3C support, which is illustrated in

Fig. 4(a). At time zero, a job of task B is released and begins executing. Shortly thereafter, a job of L3C task A is released. Since it has lower priority, it is not dispatched until all higher-priority jobs complete at time 6. Due to this accumulated interference, task A incurs excessive latency and thus misses its deadline at time 5.5. This highlights that low-priority tasks are subject to interference from higher-priority tasks within both TP_0 (task C) and any TP_X (task B) — criticality-monotonic scheduling is not L3C compatible.

In contrast, the other L3C task, task C, is given a higher priority than task B, which allows it to run with minimal latency when it arrives at time 3. However, the tradeoff is that the high-criticality jobs of task B do not enjoy freedom from unbounded interference w.r.t. task C. This is problematic: starting at time 8, a series of bursty job releases of task C cause task B to incur undue interference and to ultimately miss its deadline at time 16. With the existing PikeOS scheduling architecture, this workload cannot be supported.

Fig. 4(b) shows how the scenario plays out with the proposed approach. In this case, tasks A and C are encapsulated within rate-limited *sporadic servers* [38, 40] with a period of 10 ms and a budget of 1 ms and 2 ms, respectively, and both are assigned to an EDF band above the priority of the high-criticality task B. The blue lines show the budget of the reservation over time. For example, it can be seen that the deferrable server for task A depletes 1 ms of its 2 ms budget between times 0.5 and 1.5 as the job executes for a 1 ms duration. At time 10.5, it is replenished to its full amount.

Since L3C tasks are safely encapsulated within reservations in TP_0, they can be given higher priorities than the high-criticality task B without risking unbounded interference. This has the benefit of greatly improved latency for task A: when task A is activated at time 0.5, it is able to immediately execute, this time successfully completing before its deadline at time 5.5.

Crucially, the interference from both tasks A and C on task B is bounded due to the reservations' limited budget. That is, reservations prevent sudden bursty job releases from negatively impacting high-criticality tasks: when a job of task C arrives at time 10, it is unable to proceed until time 14 since the reservation's budget was depleted already at time 4 and is replenished only at time 13.

This example highlights how encapsulating L3C tasks within budget-constrained hard reservations allows them to meet their latency goals (when operating normally, *i.e.*, when not overloaded) while simultaneously protecting lower-priority, higher-criticality jobs from unbounded interference.

Adherence to design constraints. We briefly revisit the design constraints from Sec. 3 in light of the proposed solutions. Design constraint A (opt-in only) is satisfied since customers can simply choose to not designate any priority levels as EDF bands (which is the default). As already mentioned, design constraint B (minimally intrusive) is satisfied because the core scheduling infrastructure remains largely unchanged in place, with changes primarily related to integrating a plugin interface and delegating some queueing decisions to a reservation policy that is realized as a plugin.

Runtime efficiency (constraint C) is achieved because overheads are incurred only when EDF bands are enabled (in addition to minor plugin support costs, discussed below in Sec. 4.2), and the cost of (core-local) EDF scheduling is relatively small by modern standards.

The approach also satisfies constraint D (static configurability) since EDF bands, all reservations, and all needed resources can be allocated and configured statically, and presented to the system integrator in the usual environment. To application developers, reservations can be presented as a "dynamic part" of a TP_X, to which certain threads can be assigned *i.e.*, the fact that reservations are actually managed in TP_0 does not have to be exposed.

Constraints E (analyzability) and F (established techniques) are satisfied since reservation-based scheduling is a well-studied and widely-known approach. The proposed system can be analyzed based on a combination of existing techniques [20, 28, 39, 45, 46]; a discussion of these techniques is however beyond the scope of this paper, which is focused on the design and implementation of the system. In future work, it would be desirable to verify such an analysis as part of the Prosa project [7] to support the certification process with a firm formal foundation.

Constraint G (low barrier to entry) is satisfied since the placement of threads in reservations is transparent to application developers (*i.e.*, no special support needs to be implemented by the tasks themselves), and system integrators can be provided with *automatic* analysis support (*i.e.*, TP_X parameters can be automatically adjusted to account for bounded interference from L3C reservations) in the integration toolchain due to the large body of supporting theory for reservation-based scheduling. Further, for tasks situated below an EDF band, the fact that (some) higher-priority tasks are arbitrated in EDF order (rather than FIFO order) is immaterial [20], and tasks situated above an EDF band are still not affected by lower-priority tasks, so it is possible to gradually phase-in L3C support into an existing system with relatively minor, localized changes.

Finally, strict freedom-from-interference w.r.t. L3C tasks can be guaranteed to tasks with priorities higher than any EDF band, which satisfies constraint H, and customer-specific L3C policies (constraint I) can be supported thanks to the plugin-based implementation, which we discuss next.

4.2 Certification-Friendly Kernel Extensions

To realize the proposed design, we first implemented a plugin framework with a well-defined, generic callback API that enables the implementation of scheduler extensions. The reason for adopting this approach, instead of simply incorporating our changes into the main scheduler in PikeOS, has to do with design constraints A, B, and C. By first implementing an efficient (design constraint C), well-defined callback API upon which to build our reservation-based EDF scheduler, we allow for a static "core" of the kernel that does not change. This way, the high cost of certifying this core once can be amortized across many customers (who can reuse the certification package). If a customer requires a custom L3C policy, they only have to bear the full cost of certifying the plugin that hooks into the callback API, without having to redo the certification effort from scratch for the (much larger) core.

The plugin framework is implemented as a set of callbacks placed at strategic points in the PikeOS scheduler, thus allowing a plugin to inject scheduling logic into the control path of the main fixed-priority scheduler in response to scheduler-related events (*e.g.*, threads waking up and blocking, scheduler invocations, and scheduler timeouts). We implemented four broad categories of callbacks: *(i)* scheduling-related callbacks (*e.g.*, for dispatching or re-queueing threads), *(ii)* timer-related callbacks (*e.g.*, to handle budget-enforcement timers and job releases), *(iii)* per-thread callbacks (*e.g.*, for handling thread-related events such as waking up and blocking), and finally *(iv)* callbacks for thread admission.

The callback interface was carefully designed to not require access to core scheduler data structures from within plugins. For example, per-thread callbacks such as those for threads waking and blocking, which typically require modification of the scheduler bitmap directly, instead simply instruct the plugin framework to do so on its behalf. This separation of access is a crucial aspect of the design that ensures that plugins do not directly manipulate scheduler-private data structures, which greatly helps to make the scheduler core amenable to certification without *a priori* knowledge of the plugin used. It also makes it possible to provide binary-only distributions of the system to customers (*i.e.*, licenses that provide access only to object files without source code) without limiting the customer's ability to employ custom plugins.

The implementation makes use of the PikeOS *poke interface* [43], a mechanism to allow loadable kernel drivers to provide, apart from the regular I/O functionality, special extension functionality to the kernel binary (*e.g.*, trace points, monitoring, and security loggers). The advantage of using the poke mechanism is that if no scheduler plugin is linked into the kernel, the PikeOS scheduler works with its default implementation with negligible overhead (as the poke mechanism has been optimized to be extremely lightweight when disabled). On the other hand, when the plugin is enabled, callbacks are bound only once (during initialization), and following this, the only added overhead is that of two additional conditionals (*e.g.*, under the x86 architecture, the poke interface incurs an overhead of only three additional instructions).

An alternative approach with possibly slightly lower overheads would be a binary re-writing scheme similar to DTrace [17] or Feather-Trace [4], in which the kernel text is directly altered at runtime. However, such approaches were considered too difficult to certify as it effectively introduces self-modifying code, which is conceptually undesirable when making a safety or security case.

5 EVALUATION

The proposed design for L3C support in PikeOS, as described in the preceding sections, was implemented as a fully functional engineering prototype on PikeOS 4.1. This prototype has undergone an internal evaluation to ascertain the basic feasibility and the potential of the approach. However, due to restrictions arising from the proprietary nature of PikeOS, it is not possible to share this implementation of the proposed design. In the interest of reproducibility, we reimplemented the discussed scheduling architecture in the open-source LITMUS^{RT} research OS. All data reported in the following stems from this open-source reimplementation.[4]

We conducted experiments to validate two key claims: *(i)* under the default PikeOS scheduling architecture, it is resource-inefficient to simultaneously support low (sub-millisecond) latencies for sporadically arriving low-criticality tasks while guaranteeing isolation and acceptable performance for high-criticality tasks (as discussed in Sec. 2), and *(ii)* our proposed reservation-based EDF scheduling architecture enables L3C tasks to be safely placed in TP_0 at any priority without compromising the isolation of high-criticality tasks (*i.e.*, while introducing only a bounded amount of interference).

[4]All code can be found at https://people.mpi-sws.org/~bbb/papers/details/rtns17

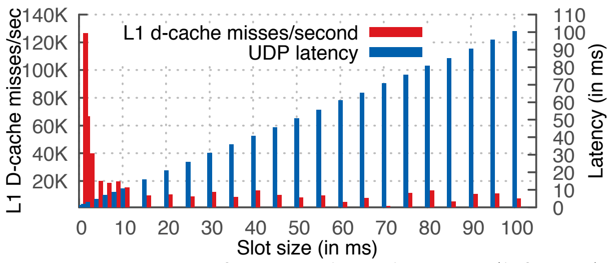


Figure 5: Latency of L3C task within TP_X (left axis) vs L1 data-cache (LOAD+STORE) misses/second for matrix-multiplication task in alternating TP_X (right axis).

Experimental setup. We re-implemented the PikeOS scheduling architecture and reservation-based L3C support in LITMUS^{RT} 2017.1 (based on Linux 4.9.30). All experiments were run on a Raspberry Pi 3b board comprising a BCM2837 SoC with a quad-core ARM Cortex-A53 CPU running at 1.2GHz, and 1 GiB of RAM. In our experiments, we used only a single core to provision real-time tasks since the focus of this paper is on intra-core interference.

We devised a synthetic mixed-criticality system configuration inspired by the use-cases described in Sec. 2.1. Our synthetic system comprised of two time partitions, each with 50% utilization and a period of 40 ms, alternating in 20 ms slots. To emulate a CPU- and cache-intensive task, the first time partition contained a matrix-multiplication task that multiplied random 10x10 matrices in a tight loop. The second time partition contained a periodic (implicit-deadline) real-time task with a WCET of roughly 10 ms and a period of 40 ms. Job releases were synchronized with partition switches. In TP_0, we emulated the presence of a high-criticality, low-utilization, low-latency real-time task (*e.g.*, a system event handler) by introducing a task with a WCET of roughly 2 ms, a period of 100 ms, and a relative (constrained) deadline of 5 ms. Finally, we emulated a network-facing L3C task using a UDP "echo" server. A UDP client, running on a separate machine, sent 128-byte UDP packets to the server, which responded with a SHA1 hash of the message.

Recall from Sec. 2 that in the stock PikeOS scheduling architecture, L3C threads may be provisioned in three different ways: *(i)* within an appropriately dimensioned TP_X, *(ii)* within TP_0 at a low priority as dictated by a criticality-monotonic priority assignment scheme, or *(iii)* within TP_0 at a high priority. Additionally, the proposed approach allows L3C tasks to be placed in reservations in TP_0. We tried each possibility to experimentally confirm the effects, tradeoffs, and benefits predicted in Secs. 2 and 4.

L3C Tasks in a TP_X. Placing an L3C task in an appropriately-dimensioned low-criticality TP_X is convenient from a certification standpoint: due to the time-partitioned scheduler, the L3C task is completely isolated and its impact on other time partitions can be determined *a priori*. Unfortunately, to guarantee low latency to sporadically arriving jobs, *frequent, small scheduling slots* are required in the time-partition schedule, which, due to frequent preemptions, negatively affects the performance of other tasks.

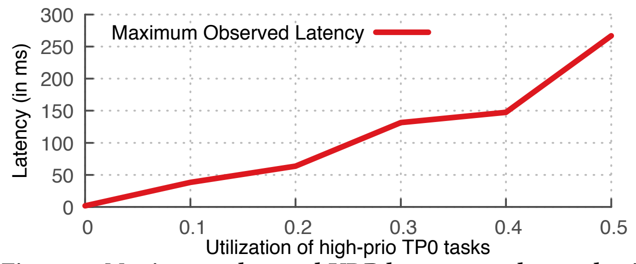


Figure 6: Maximum-observed UDP latency vs. the total utilization of higher-priority RT tasks.

To illustrate this tradeoff, we created two TP_X's with 50% utilization each, one hosting the matrix multiplication loop and the other the (L3C) UDP echo server. We varied the slot size of the partitions while maintaining the same utilization (*i.e.*, the scheduling table always comprised of two slots of equal size, one for each TP_X).

We measured the round-trip latency of packets sent to the UDP server and the L1 data-cache misses of the matrix-multiplication task. Cache misses were counted with the Cortex-A53's built-in *performance monitoring unit* (PMU). Fig. 5 depicts the results.

First, as expected, the latency of the L3C task reduces linearly as the slot size is decreased. This is unsurprising as the slot size determines the worst-case scheduling delay (*i.e.*, the interval during which the L3C task has pending work but is not scheduled because a different TP_X is currently active). However, as can be seen in the figure, in order to achieve latencies below 10 ms,[5] the small slot sizes result in a significant increase in the number of cache misses of the matrix multiplication task. For a slot size of 5 ms, we observed around 18,000 cache misses per second, compared with a median of around 9,300 cache misses for all measured slot sizes above 15 ms (measured in incremental steps of 5 ms, up to 100 ms). Further, attempting to support sub-millisecond latencies results in a further 3x increase in the number of cache misses (around 66,000 misses per second with a 1 ms slot size).

To summarize, lowering latency by placing L3C tasks within a frequently activated TP_X results in substantially increased system overheads, which negatively impacts the performance of other tasks in the system. Compensating this effect can, for example, necessitate substantially larger resource allocations to other tasks, or a switch to more capable hardware, which is undesirable. If a low-criticality task requires "low" latency, which on this hardware means less than about 10 ms, then placing it in a TP_X is problematic.

L3C tasks in TP_0 at a low priority. When placing L3C tasks in TP_0 under a criticality-monotonic priority-assignment scheme, freedom-from-interference can be guaranteed for higher-criticality (and hence higher-priority) tasks. Unfortunately, this also means that they in turn interfere with the L3C task, thus increasing its latency.

To quantify this effect, we placed the (L3C) UDP echo server in TP_0 at a low priority and created a number of higher-priority real-time tasks (also within TP_0) and varied their total utilization from zero (*i.e.*, no higher-priority tasks were present) to 0.5 (*i.e.*, higher-priority tasks use up 50% of the execution time on the CPU). In our setup, based on the workload reported in an automotive benchmark provided by Bosch [22], we spawned seven higher-priority tasks using harmonic task periods ranging from 5 ms to 1000 ms. Each task was assigned $\frac{1}{7}$ of the total utilization and the per-job execution cost was determined based on each task's period.

[5]Studies have shown that for interactive digital interfaces there is a perceivable improvement in user experience with latencies well below 10 ms [30].

Fig. 6 shows the results of the experiment. The maximum observed UDP latency can be seen to increase significantly as the utilization of the high-priority tasks in TP_0 increases; when higher-priority tasks have a utilization of 50%, the latency increases to over 250 ms, compared with around 1.2 ms under no load.

While this result is not surprising *per se*, it does demonstrate that the additional interference from all higher-criticality tasks in the system can result in *substantial* latency in low-criticality tasks. Note that *any higher-criticality tasks*, even those within a TP_X, may potentially interfere with the L3C task in TP_0. In particular, a criticality-monotonic priority assignment scheme may result in a lower priority for the L3C task in TP_0 compared to placing it within a TP_X, incurring further interference from tasks whose priority would be relatively lower otherwise.

L3C tasks in TP_0 at a high priority. An alternative to the criticality-monotonic priority-assignment scheme, we placed the UDP echo server placed at the highest priority within TP_0. Under this setup, we observed average latencies of only 1.35 ms and a maximum latency of 1.57 ms at a rate of ten requests per second. However, this approach unsurprisingly exposes lower-priority (but potentially higher-criticality) tasks to possibly unbounded interference.

To illustrate the danger of such interference, we measured the response times of high-criticality real-time task under three scenarios. Recall that, in our experimental setup, one of the two time partitions contains a periodic, implicit-deadline real-time task with a budget of 10 ms and a period of 40 ms (henceforth "task A"). There is further a high-criticality system event handler in TP_0 with a budget of 2 ms, period of 100 ms, and a relative deadline of 5 ms ("task B"). We measured both tasks' response times in the presence of the L3C UDP server with *(i)* no load, *(ii)* with a "normal" rate of ten requests per second, and finally *(iii)* with a baseline rate of ten requests per second and sporadic one-second bursts of 1,000 requests per second. Response times were measured with LITMUS^{RT}'s schedule tracing facility over the course of 60 seconds.

Fig. 7 and Fig. 8 show the results for tasks A and B, respectively. (Ignore for now the curve labeled "Bursty w/ res," which corresponds to the proposed reservation-based scheme and is discussed below.) First note that the curves corresponding to the no-load and normal-load scenarios overlap, which indicates that the normal load imposes only negligible interference. However, in the bursty scenario, prominent spikes are visible in both Figs. 7 and 8. In the case of task A (Fig. 7), response times can be seen to increase 40x from around 10 ms to over 400 ms. Similarly, the response times of task B (Fig. 8) rise from below 2 ms to 6 ms. Both tasks A and B miss their deadlines of 40 ms and 5 ms, respectively.

Crucially, this experiment shows that lower-priority (both high- and low-criticality) tasks do not have freedom from interference from higher-priority L3C tasks in TP_0. In particular, the OS cannot guarantee *bounded* interference, which makes certification problematic. For example, in our example, the UDP server, if placed at maximum priority, might implement a custom rate-limiting solution, the correctness of which would have to be certified at the level of assurance of the highest-criticality real-time task in the system.

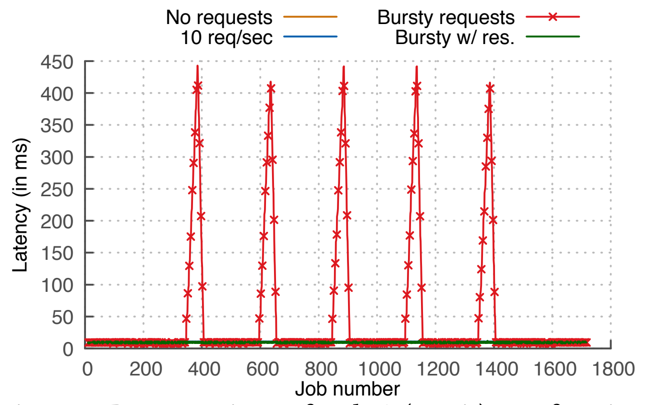


Figure 7: Response times of task A (Y axis) as a function of job number (X axis) under four different scenarios. All curves overlap, with exception of the promiment spikes of the curve labeled "Bursty requests."

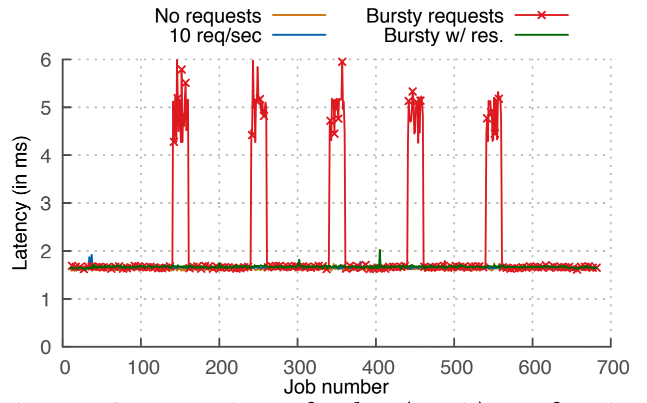


Figure 8: Response times of task B (Y axis) as a function of job number (X axis) under four different scenarios. All curves overlap, with exception of the promiment spikes of the curve labeled "Bursty requests."

In conclusion, our results demonstrate that placing L3C tasks in either a TP_X or within TP_0 (regardless of the priority at which it is placed) leads to unsatisfactory, resource-intensive, or unsafe results. We now show how the proposed approach allows the L3C thread to be safely placed at maximum priority within TP_0.

L3C tasks in reservations within TP_0. We placed the L3C task in TP_0 within a maximum-priority, but rate-limited reservation with a budget of 1 ms and a period of 50 ms (*i.e.*, provisioned for roughly 20 requests per second) and reran the previous experiment (with burst-induced transient overloads of the UDP echo server).

The curves labeled "Bursty w/ res" in Figs. 7 and 8 show the resulting response times of tasks A and B, respectively. As can be inferred from the fact that the curves overlap completely with those corresponding to the no-load and normal-load scenarios, there are no significant changes in the response times of tasks A or B during request-rate spikes: as expected, temporal isolation is ensured. At the same time, the observed UDP latency under normal load is just as low as before, which shows the that proposed approach succeeds in accommodating the needs of L3C tasks without jeopardizing high-criticality tasks, the goal of our case study.

6 RELATED WORK

Mixed-criticality systems. Following the now-famous MCS paper by Vestal [44], there has been a tremendous amount of research in this field, with a strong focus on the theoretical underpinnings of specialized scheduling strategies; we refer the interested reader to Burns and Davis' excellent survey [5].

The bulk of the mixed-criticality literature has focused on ensuring guarantees for high-criticality tasks in both normal and degraded modes, with low-criticality tasks receiving only best-effort or no service once the system enters degraded mode. However, some approaches based on slack distribution are aimed at improving the performance of low-criticality tasks by allowing low-criticality tasks to use the slack available in the schedule of high-criticality tasks [9, 10, 18, 19, 27, 31].

In particular, Groesbrink *et al.* [18] approach the problem explicitly from a certification point-of-view and guarantee a minimum bandwidth for VMs running on a real-time hypervisor by employing periodic reservations and an elastic task model. A mode-change triggers resource redistribution, which reassigns spare capacity.

While these approaches are suitable for a clean-slate scheduler implementation, they are hard to integrate into an existing kernel like PikeOS without violating one or more of the constraints described in Sec. 3. However, we note that other implementations of mixed-criticality schedulers have successfully applied such techniques, including one by Herman *et al.* [21] and the XtratuM hypervisor for the avionics domain [6, 8].

Reservations and hierarchical scheduling. Processor reservation are a classic technique by which one or more levels of indirection are introduced in the real-time scheduler, that is via a (multi-level) *hierarchical scheduling* scheme, to separate high-level real-time policy from low-level process dispatching. In this hierarchy of schedulers, parent schedulers allocate CPU time to one of their child schedulers, and this process is repeated until a leaf node (task) is chosen. The advantage of hierarchical scheduling is that it is amenable to compositional analysis, and there is a large body of real-time literature dedicated to this topic [1, 2, 11, 12, 14, 23–26, 29, 32–34, 36–38, 42]. In this paper, we simply employ existing real-time server algorithms in the support of L3C tasks.

In PikeOS, we chose to implement deferrable servers [24, 42] due to their simplicity, although it would be interesting to compare different reservation schemes (*e.g.*, sporadic servers [38, 41] and constant-bandwidth or CBS servers [1]) from a certification point-of-view under consideration of the constraints described in Sec. 3.

EDF within fixed-priorities. The idea of performing EDF scheduling within a fixed-priority scheduler is not a new one [20, 28]. In particular, Harbour and Palencia [20] presented a response-time analysis for "EDF within priorities", a model very similar to ours where several EDF priority levels may be designated and EDF scheduling is used for tasks within those priority levels.

Slot shifting. Finally, another approach that could be used to support L3C tasks is slot shifting [15, 16, 35], where the static partition schedule is dynamically altered at runtime when sporadic jobs are released by redistributing statically determined spare capacity (*i.e.*, by shifting periodic workloads without violating their deadlines). While slot shifting is a well-established technique, in the context of PikeOS, we decided against the use of slot shifting as it requires runtime changes to the static schedule, which is potentially problematic from a certification point-of-view, and because it represents a more substantial departure from the established PikeOS scheduling model (especially w.r.t. constraints B, C, G, and H in Sec. 3).

7 CONCLUSION

We conducted a case study exploring OS-level support for L3C tasks in practical mixed-criticality systems, and presented the design challenges involved in implementing such support in PikeOS, a certified real-time OS that has seen widespread use in the real world. Under our proposed solution, L3C tasks are encapsulated within reservations scheduled via an EDF scheduler, thus allowing L3C tasks with low-latency requirements to be safely placed within TP_0 at designated EDF priority levels. We presented the design and implementation of our solution in PikeOS as well as results from our evaluation of an alternate open-source re-implementation in LITMUS^{RT} on a 64-bit ARM SoC. Our results show that the current PikeOS scheduler design cannot simultaneously realize low latencies for L3C tasks, low partition-switching overheads, and bounded interference for high-criticality tasks. In contrast, our approach leverages reservations to guarantee, with minimally invasive changes, freedom from *unbounded* interference, as well as freedom from *any* interference (for tasks prioritized above any EDF band), while enabling low latencies with low overheads. In conclusion, while supporting demanding L3C tasks with stringent latency constraints — in particular, supporting them *well* — is a challenge for static time-partitioned systems, the problem can be effectively and efficiently solved with a careful, certification-cognizant integration of state-of-the-art reservation techniques.

Acknowledgement. This work has been supported by grant 621429 as part of the European Union's EMC^2 program.

REFERENCES

[1] Luca Abeni and Giorgio Buttazzo. 1998. Integrating multimedia applications in hard real-time systems. In *RTSS 1998*. IEEE.
[2] Luis Almeida and Paulo Pedreiras. 2004. Scheduling within temporal partitions: response-time analysis and server design. In *ICES 2004*. ACM.
[3] Sanjoy K Baruah, Alan Burns, and Robert I Davis. 2011. Response-time analysis for mixed criticality systems. In *RTSS 2011*. IEEE.
[4] B Brandenburg and J Anderson. 2007. Feather-Trace: A lightweight event tracing toolkit. In *OSPERT 2007*.
[5] Alan Burns and Robert Davis. 2017. *Mixed criticality systems—a review, 9th edition.* Technical Report. Department of Computer Science, University of York.
[6] E Carrascosa, Javier Coronel, Miguel Masmano, Patricia Balbastre, and Alfons Crespo. 2014. XtratuM hypervisor redesign for LEON4 multicore processor. *ACM SIGBED Review* 11, 2 (2014), 27–31.
[7] Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. 2016. PROSA: A Case for Readable Mechanized Schedulability Analysis. In *ECRTS 2016*.
[8] Alfons Crespo, Ismael Ripoll, and Miguel Masmano. 2010. Partitioned embedded architecture based on hypervisor: The xtratum approach. In *EDCC 2010*.
[9] Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. 2009. On the scheduling of mixed-criticality real-time task sets. In *RTSS 2009*.
[10] Dionisio De Niz, Lutz Wrage, Anthony Rowe, and Ragunathan Raj Rajkumar. 2014. Utility-based resource overbooking for cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 5s (2014), 162.
[11] Zhong Deng and JW-S Liu. 1997. Scheduling real-time applications in an open environment. In *RTSS 1997*.
[12] Arvind Easwaran, Madhukar Anand, and Insup Lee. 2007. Compositional analysis framework using EDP resource models. In *RTSS 2007*.
[13] Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, and Eduardo Tovar. 2015. How realistic is the mixed-criticality real-time system model?. In *RTNS 2015*.
[14] Xiang Feng and Aloysius K Mok. 2002. A model of hierarchical real-time virtual resources. In *RTSS 2002*.
[15] Gerhard Fohler. 1995. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *RTSS 1995*.
[16] Gerhard J Fohler. 1994. *Flexibility in statically scheduled hard real-time systems.*
[17] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD.* Prentice Hall Professional.
[18] Stefan Groesbrink, Luis Almeida, Mario de Sousa, and Stefan M Petters. 2014. Towards certifiable adaptive reservations for hypervisor-based virtualization. In *RTAS 2014*.
[19] Fei Guan, Long Peng, Luc Perneel, Hasan Fayyad-Kazan, and Martin Timmerman. 2017. A Design That Incorporates Adaptive Reservation into Mixed-Criticality Systems. *Scientific Programming* 2017 (2017).
[20] M Gonzales Harbour and J Carlos Palencia. 2003. Response time analysis for tasks scheduled under EDF within fixed priorities. In *RTSS 2003*.
[21] Jonathan L Herman, Christopher J Kenna, Malcolm S Mollison, James H Anderson, and Daniel M Johnson. 2012. RTOS support for multicore mixed-criticality systems. In *RTAS 2012*.
[22] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. 2015. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.
[23] Tei-Wei Kuo and Ching-Hui Li. 1999. A fixed-priority-driven open environment for real-time applications. In *RTSS 1999*.
[24] John P Lehoczky. 1987. Enhanced aperiodic responsiveness in hard real-time environments. In *RTSS 1987*.
[25] Giuseppe Lipari and Sanjoy Baruah. 2001. A hierarchical extension to the constant bandwidth server framework. In *RTAS 2001*.
[26] Giuseppe Lipari and Enrico Bini. 2003. Resource partitioning among real-time applications. In *ECRTS 2003*.
[27] Giuseppe Lipari and G Buttazzo. 2013. Resource reservation for mixed criticality systems. In *Proceeding of Workshop on Real-Time Systems: the past, the present, and the future*. 60–74.
[28] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.
[29] Clifford W Mercer, Stefan Savage, and Hideyuki Tokuda. 1993. Processor capacity reserves: An abstraction for managing processor usage. In *Workstation Operating Systems*.
[30] Albert Ng, Julian Lepinski, Daniel Wigdor, Steven Sanders, and Paul Dietz. 2012. Designing for low-latency direct-touch input. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*.
[31] Taeju Park and Soontae Kim. 2011. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *ICES 2011*.
[32] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. 1997. Resource kernels: a resource-centric approach to real-time and multimedia systems. In *Multimedia Computing and Networking 1998*, Vol. 3310. 150–164.
[33] John Regehr and John A Stankovic. 2001. HLS: A framework for composing soft real-time schedulers. In *RTSS 2001*.
[34] Saowanee Saewong, Ragunathan Rajkumar, John P Lehoczky, and Mark H Klein. 2002. Analysis of Hierarchical Fixed-Priority Scheduling.. In *ECRTS 2002*.
[35] Stefan Schorr and Gerhard Fohler. 2013. Integrated time-and event-triggered scheduling-An overhead analysis on the ARM architecture. In *RTCSA 2013*.
[36] Insik Shin and Insup Lee. 2003. Periodic resource model for compositional real-time guarantees. In *RTSS 2003*.
[37] Insik Shin and Insup Lee. 2008. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 30.
[38] Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems* 1, 1 (1989), 27–60.
[39] Marco Spuri. 1996. *Analysis of deadline scheduled real-time systems.* Ph.D. Dissertation. Inria.
[40] Marco Spuri and Giorgio Buttazzo. 1996. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems* 10, 2 (1996), 179–210.
[41] Mark Stanovich, Theodore P Baker, An-I Wang, and Michael Gonzalez Harbour. 2010. Defects of the POSIX sporadic server and how to correct them. In *RTAS 2010*.
[42] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.* 44, 1 (1995), 73–91.
[43] Henrik Theiling. 2016. *'Poke' — A Fast, Generic, Low-Level Dispatch Mechanism for PikeOS Kernel Drivers.* Technical Report. SYSGO AG.
[44] Steve Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS 2007*.
[45] Fengxiang Zhang. 2009. *Analysis for EDF scheduled real-time systems.* Department for Computer Science, University of York.
[46] Fengxiang Zhang and Alan Burns. 2009. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans. Comput.* 58, 9 (2009), 1250–1258.