TimerShield: Protecting High-Priority Tasks from Low-Priority Timer Interference

Pratyush Patel Manohar Vanga Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

***Abstract*—Timer interference arises when a high-priority real-time task is delayed by a timer interrupt that is intended for a lower-priority task. We demonstrate that high-resolution timers, as exposed for instance by Linux's *hrtimer* API, can cause substantial timer interference, which manifests as significantly increased response times and lowered throughput. To eliminate this source of unpredictability, we propose TimerShield, a priority-aware high-resolution timer subsystem that selectively delays the servicing of lower-priority timer interrupts while a high-priority task is executing. We present the design and implementation of a fully functional TimerShield prototype in Linux PREEMPT_RT and compare it against Linux's stock *hrtimer* subsystem on two different platforms (x86 and ARM). Our results show that TimerShield adds only little overhead, while completely eliminating the timing unpredictability and throughput degradation caused by unnecessary interrupts.**

I. Introduction

Modern real-time operating systems make use of high-precision timing within various subsystems; these include the implementation of various scheduler features (*e.g.*, accurate budget enforcement), as well as providing high-resolution self-suspension calls to userspace processes. For example, POSIX's `clock_nanosleep()`, which blocks a process for a specified number of nanoseconds, finds its use in accurately timing periodic job releases and time-critical control loops.

These features have grown more precise over the years thanks to the introduction of dedicated, high-resolution, one-shot timer hardware into modern CPUs,[1] and operating systems such as Linux provide dedicated subsystems to work with these devices.

An example of this is the *hrtimer* kernel subsystem in Linux, which is responsible for providing an API for managing software timers and multiplexing them on these high-precision, one-shot hardware timers [13]. The *hrtimer* subsystem maintains the set of all pending software timers in the system. When a software timer expires, it re-programs the timer hardware to raise an interrupt at the next-earliest timer expiry based on the current set of pending software timers.

However, due to the way interrupt hardware works, interrupts generated by these high-resolution timers effectively execute with the highest priority in the system, regardless of the priority of the process that created the interrupt-causing timer. As a consequence, interrupts generated by timers of low-priority tasks can end up preempting high-priority processes. While the overhead incurred from a single timer may seem relatively small, the overhead from a large number of timers can quickly add up. This is further exacerbated by current POSIX-based operating systems, which do not impose any strict limits on which task can create timers and how many they may create.

[1]For example, both 32-bit and 64-bit Intel machines, and many ARM SoCs provide such dedicated hardware (via the Local Advanced Programmable Interrupt Controller or LAPIC [17] and the High Precision Event Timer (HPET) [16] on x86, and via a core-local, high-resolution timer on ARM).

This is problematic for two reasons: from an analysis point of view, accounting for such interference results in severe pessimism as any process in the system can create timers and cause preemptions in higher-priority tasks. From a systems point of view, constant preemptions result in increased worst-case response times and lowered throughput for high-priority tasks, as well as increased unpredictability and jitter.

However, servicing timer interrupts corresponding to timers of tasks with lower priority than the currently executing task is a complete waste of CPU cycles. Even if a low-priority timer interrupt is serviced immediately, the task that programmed it cannot react to the timer expiration while runnable higher-priority tasks exist in the scheduler's ready-queue. A better approach would be to defer the servicing of the low-priority timer until all higher-priority tasks have finished executing, thereby avoiding the unnecessary preemption of higher-priority tasks.

Based on the above observation, in this paper we present the design of the TimerShield subsystem, an interference-free, high-resolution timer subsystem for fixed-priority RTOSes. TimerShield provides all the same timer management primitives as Linux's *hrtimer* subsystem (creation, deletion, and expiration of timers). However, TimerShield additionally introduces the notion of priority to timers set by userspace processes. Thus, under TimerShield, whenever a process at a particular priority is scheduled, all timers of lower priority processes are automatically "masked" before determining the earliest time with which to reprogram the timer hardware, preventing them from preempting the higher-priority process. As this priority-aware masking and earliest-timer lookup is on the context-switch hotpath, TimerShield makes use of specialized data structures to minimize the added runtime overhead.

Contributions. This paper makes the following contributions.

- We illustrate the problem of interrupt interference in the current design of the high-resolution timer subsystem in Linux (Sec. IV).
- We propose TimerShield, a high-resolution timer subsystem design that eliminates lower-priority timer interference in such kernels with negligible overhead (Sec. III).
- We describe our implementation of TimerShield in Linux (with the PREEMPT_RT real-time patch) (Sec. III).
- We evaluate and compare TimerShield with the stock *hrtimer* subsystem in Linux on two hardware platforms

(x86 and ARM). Our results show that in relation to the significant benefit of avoiding interrupt interference, the tradeoff of using a more specialized data-structure with moderately higher overheads is justified (Sec. IV).

II. Background

We begin by providing some background on how timers work in Linux. However, we note that the design of TimerShield is not Linux-specific and that it can be adapted to other OSes easily.

A. Timer Hardware and Interrupt Handling in Linux

In modern Intel x86 CPUs, each core has a high-resolution, on-die local APIC (LAPIC) timer that is directly connected. As a result, access to the LAPIC timer is significantly faster than to previously available timer hardware (*e.g.*, the older *Programmable Interval Timer* (PIT) is external to the CPU and thus has significantly slower access times). Similarly, recent ARM processors have a per-core, high-resolution timer. Operating systems typically configure these timers in one-shot mode, where a specified counter is decremented every cycle and a timer interrupt is raised when the count reaches zero.

Linux (PREEMPT_RT) handles such interrupts via a split-handler model, where the interrupt handling code is split into a short, time-critical *top half*, and a longer *bottom half* [6, 20]. The top half runs as soon as the interrupt signal is received by preempting the currently executing task, performs the absolutely critical work such as acknowledging interrupts to the hardware, and defers the rest of the interrupt handling to the bottom half, which is scheduled for a later point in time. Subsystems may choose to use either of two methods for handling high-resolution timer interrupts: handle everything within the top half (used for time-critical "wakeup"-related interrupts such as those for `clock_nanosleep()`), or defer a bulk of the work to the bottom half (used for less critical "signaling"-based timer interrupts, such as the *timerfd* interface that allows processes to be notified of timer events via file descriptors). The downside of the latter approach is increased interrupt latency as the bottom half is subject to scheduling and queueing delays. We now present a broad overview of the Linux timer subsystems before explaining its *hrtimer* subsystem in detail.

B. Linux Timer Subsystems

The application of timers in Linux falls broadly into two categories: *(i) high-resolution timers* that are used for high-precision timed events (*e.g.*, `clock_nanosleep()`), and *(ii) low-resolution timers* that are used for timed events that do not require high precision (*e.g.*, timeouts for I/O requests that signal error conditions). In Linux, low-resolution timers are handled via the *timer wheel* subsystem [41], while high-resolution timers are handled by the *hrtimer* subsystem, both of which are higher-level APIs built on top of the fundamental timer-handling architecture described in Sec. II-A.

The timer wheel attempts to strike a balance between precision and efficiency and supports fast $O(1)$ insertion and deletion of timers, which is useful as low-resolution timers are often cancelled before they expire; for example I/O timeouts only fire under relatively rare error conditions. Unfortunately, the tight integration of the timer wheel design with the tick-based mechanism in Linux presents many challenges to using it for high-resolution timers. Supporting high-precision timers with the same design would require an increase in timer ticks per second [38] which negatively impacts application performance.

Rather, high-resolution timers are handled in Linux via the *hrtimer*[2] subsystem, which we describe next.

C. *The* hrtimer *Subsystem*

The *hrtimer* subsystem has two key roles: it provides a flexible timer-management API (*i.e.*, timer creation, reprogramming, and deletion operations), and invokes callbacks on timer expiry.

As *hrtimers* are a part of some of the kernel's hot paths such as the scheduler, its data structures are designed for efficiency. Each *hrtimer* is represented in the kernel by a structure containing, among other things, the absolute expiry time and a callback pointer. These structures are maintained in earliest-expiration-time order within a self-balancing *red-black tree* (rb-tree) that enables $O(\log n)$ insertion and deletion of timers as well as $O(1)$ access to the earliest-expiring timer (by additionally storing a pointer to the minimum element in the root node). A single rb-tree is maintained for each high-resolution timer available in hardware (typically one per core).[3]

When inserting or deleting from the rb-tree, the kernel checks if the underlying hardware requires reprogramming (*i.e.*, if a timer with an earlier expiration time than the current earliest timer is inserted, or if the timer with the earliest time is deleted).

When a timer expires, an interrupt is raised by the hardware, the timer is removed from the rb-tree, and the callback supplied within the timer is invoked before reprogramming the timer hardware with the next-earliest time (if any). If multiple timers expire at once, they are all cleared sequentially.

Recall that the *hrtimer* subsystem forms the basis for implementing the POSIX `clock_nanosleep()` system call, which is used to suspend the currently executing process for a specified number of nanoseconds or until a specified absolute point in time is reached. This is particularly relevant in real-time systems where it is used to accurately time periodic job releases, self suspensions, and control loops. However, the *hrtimer* subsystem does not consider the priorities of the processes that created the timers when reprogramming the underlying hardware. Consequently, the top half of high-resolution timers *always* cause preemptions in high-priority tasks even if the timer was created by a lower-priority task. We describe next how TimerShield efficiently avoids this unnecessary interference.

III. TimerShield: Design & Implementation

TimerShield extends the *hrtimer* architecture in three ways. First, it extends the data structures such that, for every timer,

[2]The term *hrtimer* may be misleading because Linux does not guarantee that the timers managed by this subsystem will have high resolution. It simply provides a clean interface to deal with such timers and whether they support nanosecond-level precision depends on the underlying hardware. In this paper, we assume the availability of high-resolution timer hardware.

[3]A single rb-tree is actually maintained per "clock base," and multiple clock bases may be multiplexed on the same timer hardware, but for simplicity we assume one rb-tree per hardware timer in this paper.

it keeps track of the priority of the process that created it (Sec. III-A). Second, it introduces a new interface that enables efficient, priority-aware queries of the earliest-expiring timer (Sec. III-B). Third, it modifies the timer-expiration logic so that only those expired timers with a priority greater than or equal to that of the current process are serviced (Sec. III-C).

With this infrastructure in place, TimerShield intelligently reprograms the timer hardware (on each core) at the end of each context switch and timer expiry with the expiration time of the earliest-expiring timer with a priority greater than or equal to that of the process about to be scheduled. This eliminates timer interference without affecting the timing of lower-priority tasks.

In the following, we discuss the design and implementation of TimerShield, starting with the structural changes that were needed to make the timer subsystem "priority-aware."

A. Priority-Aware High-Resolution Timers

Since Linux's *hrtimer* structures do not have a notion of priority, we extended the data type with a priority field and modified the timer creation logic to store the scheduling priority of the process that is creating (or reprogramming) it.

Recall that Linux uses a single rb-tree to store all *hrtimers* of a single timer (and maintains a direct pointer to the timer with least expiration time within the root node of this rb-tree). In order to later implement priority-aware queries efficiently (Sec. III-B), we modified this single rb-tree to be an *array* of rb-trees, one per priority level. This is because, in order to determine the earliest-expiring timer *at a particular priority level*, a single rb-tree would require an $O(n)$ search over all timers enqueued in the rb-tree. With an array of rb-trees, one per priority level, this is reduced back to $O(1)$: we simply retrieve the earliest-expiring timer from the root node of the red black tree at the array index corresponding to a particular priority level. Note that the use of an array of rb-trees also maintains Linux's logarithmic time complexity for the creation and cancellation of *hrtimers*. As Linux supports 140 process priority levels, each hardware timer module has a corresponding 140-element array of rb-trees associated with it under TimerShield.

B. Priority-Aware Earliest-Expiration Queries

We now describe a new interface for determining the earliest-expiring timer with a priority greater than or equal to a given threshold. The key challenge lies in making such priority-aware queries *fast*, as they are invoked frequently in kernel hotpaths (*e.g.*, as part of each context switch and in interrupt top halves).

We found that this requirement maps directly to the well-studied *range minimum query* (RMQ) problem [5, 11]. The RMQ problem asks to find the minimum element in a sub-array of an array of comparable objects. Under TimerShield, this corresponds to the earliest-expiring timer with a priority in the range $[current\_priority, max\_priority]$, where $current\_priority$ is the priority of the currently running process, and $max\_priority$ is the highest-possible priority.

We prototyped two candidate RMQ solutions: a segment tree [34], and two binary-indexed trees [11]. We chose the former

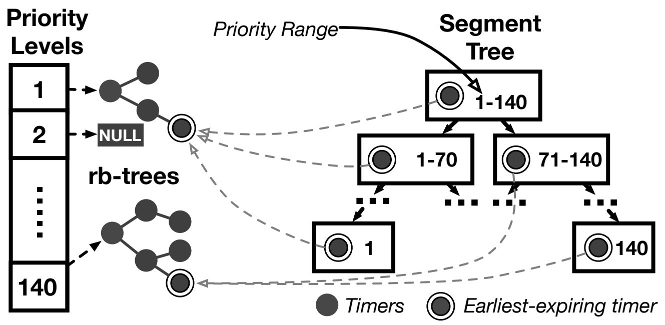

Fig. 1: The data structures used in TimerShield.

for TimerShield as it consumed less memory, updated faster, and was simpler to implement compared to the latter.

Segment trees. A segment tree is a complete binary tree, where each node represents a contiguous interval of the total search space. Each node contains the minimum value in its interval.

TimerShield uses a single segment tree where each node represents a range of priority values, and stores a pointer to the earliest-expiring timer in that priority range. As can be seen in Fig. 1, the root node points to the earliest-expiring timer in the entire range of priorities; its left child points to the earliest-expiring timer in the range $[1, max\_priority/2]$, and its right child points to the earliest-expiring timer in the range $[max\_priority/2, max\_priority]$. Subsequent levels of the tree consist of nodes representing the earliest-expiring timers from similarly half-split intervals. Finally, the leaf nodes denote the earliest timer corresponding to a single priority level which are obtained from the corresponding rb-trees. If no timers exist for that priority level, then the node pointer contains `NULL`.

The segment tree supports two basic operations: updates and queries. An update operation is performed when a leaf node changes its value (*e.g.*, the earliest-expiring timer at a priority level changed due to a timer expiration). The value at the leaf node is modified and the tree is traversed upward to the root node while re-calculating the earliest-expiring timer at each level.

A range query proceeds a in top-down fashion from the root in a manner that is most easily described recursively, although the actual implementation in TimerShield uses a faster iterative variant. When encountering a node representing a range $[a, b]$, there are three possibilities: *(i)* if $[a, b]$ lies completely in the queried subrange, then the node's value is returned immediately; *(ii)* if $[a, b]$ only partially overlaps the queried subrange, then the minimum of (recursive) range queries on the left and right subtrees of the node is returned; and otherwise, *(iii)* if there is no overlap, then the node is disregarded.

Both the update and the range-query operations visit $O(\log(max\_priority))$ nodes in the worst-case. Since the number of priority levels in Linux is fixed, and there is one leaf node in the segment tree per priority level, the size of the segment tree (which is a full binary tree) is constant and can be represented by an array of $(2 \times max\_priority - 1)$ nodes.

C. Putting It All Together

With the core TimerShield data structures in place, we next explain how TimerShield operates.

Creation and deletion of timers. To create (or reprogram) a timer, the current process' priority is retrieved from the *process control block* (PCB) and the timer is inserted into the rb-tree associated with that priority level. Deleting a timer simply involves removing it from the rb-tree. The segment tree is initialized to an array of `NULL` pointers during system startup. As timers are created and deleted, if the earliest-expiring timer at a given priority level changed, the segment tree is updated.

The overall insertion operation completes in logarithmic time as rb-tree insertion requires $O(\log(N))$ operations in the worst case, where N is the number of timers at the priority level being inserted into, and the segment-tree update (if any) requires $O(\log(max\_priority))$. Similarly, deleting a timer under TimerShield requires logarithmic time as it too involves an rb-tree deletion and a potential segment tree update.

Expiration of timers. When a timer expiration interrupt fires, TimerShield only services expired timers with a priority greater than or equal to the currently running process. Any expired lower-priority timers are ignored, avoiding unnecessary delays in high-priority process execution from the servicing of low-priority timers (which would anyway not be received until the high-priority task blocks). Finally, TimerShield reprograms the timer hardware if an expiration warranted a change.

Timer reprogramming during context switch. During timer creation, deletion, and expiration, the earliest-expiration timer is determined based on the priority of the currently executing process. However, if a context switch occurs, this priority may change. Thus, TimerShield also reprograms the timer hardware at the end of each context switch based on the *priority of the process that is about to be scheduled*. That is, a range query is performed on the segment tree to determine the earliest-expiring timer in the range $[next\_priority, max\_priority]$, where $next\_priority$ is the priority of the process about to be scheduled. If the earliest-expiring timer changed, the timer hardware is reprogrammed accordingly.

Scheduler timers. Not all timers are associated with a process priority. In particular, schedulers use *hrtimers* to implement features such as budget enforcement and rate limiting, and delaying these interrupts can result in incorrect behavior in some cases. In order to make sure that these interrupts are never masked, we modified the scheduler to create critical timers with the highest system priority to ensure their immediate delivery.

However, note that every scheduler interrupt need not be at the highest system priority. In Linux, schedulers are organized hierarchically into *scheduler classes* and traversed top-down when picking a process to schedule. Thus, a higher scheduler class can be safely assigned a higher priority than a lower scheduler class as the lower-level scheduler would not be invoked while the higher scheduler has processes to schedule. Timers belonging to the SCHED_FIFO and SCHED_RR schedulers (such as real-time bandwidth-throttling timers) are thus given the highest priority, while timers belonging to the SCHED_NORMAL scheduling class (*i.e.*, the non-real-time timesharing policy) are given a priority lower than any real-time task, but still higher than any priority available to SCHED_NORMAL processes.

Dynamic priorities. An assumption in TimerShield is that the priority of a task that creates a timer remains unchanged until the timer fires. However, even under a fixed-priority scheduling policy, the effective scheduling priority of a task may change dynamically at runtime, either implicitly due to priority inheritance, or explicitly due to task parameter changes.

With regard to priority inheritance, TimerShield transparently supports the common case, where timers are used to perform explicit sleeps *outside* of critical sections, as then critical sections (during which priority inheritance might occur) and the use of timers never coincide. Similarly, TimerShield transparently supports explicit priority changes for any task (ready or blocked on I/O) as long as the task does not have pending timers.

Our TimerShield prototype presently does not deal with cases where sleeps are performed *within* critical sections (*i.e.*, while holding a lock); we are unaware of any practical use-case for this. However, if desired, support for sleeps inside critical sections, and dynamic priority changes in general, could be added by keeping track of the timers that a process has programmed within its PCB, and by then simply restarting all of a process' pending timers whenever its priority changes.

To summarize, TimerShield extends the *hrtimer* subsystem such that low-priority software timer interrupts never preempt high-priority tasks. It achieves this by intelligently "masking" timers of lower priority than the currently executing task, and maintains low overhead through the combination of per-priority rb-trees and segment trees for efficient range-minimum queries.

IV. Evaluation

We conducted an evaluation of TimerShield on two hardware platforms to answer the following key questions: *(i)* what additional overheads does TimerShield introduce? *(ii)* How does TimerShield improve the predictability of high-priority real-time tasks? *(iii)* How does TimerShield affect lower-priority tasks?

A. Experimental Setup

We evaluated TimerShield on two platforms: x86 and ARM, and although the machines were quad-core machines, we only enabled a single core in both setups for simplicity of evaluation. However, it should be noted that TimerShield extends trivially to multi-core systems as timer hardware, software, and data structures (*e.g.*, locks) are all core-local.

Intel machine. We used a 3.2 GHz quad-core Intel Core i5-3740 CPU with 32 KiB of L1 instruction and data caches, and unified L2 and L3 caches of 256 KiB and 6 MiB respectively. Hyperthreading and power-saving features were disabled.

ARM machine. We used a Raspberry Pi 3B board with a Broadcom BCM2837 SoC comprised of a 1.2 GHz quad-core ARM Cortex-A53 CPU and 16 KiB L1 instruction and data caches as well as a unified 512 KiB L2 cache.

In both setups, we performed all experiments on Linux patched with PREEMPT_RT (version 4.6.1-rt2 for x86_64, and version 4.4.12-rt19 patched on top of the vendor-supplied rpi-4.4.14 kernel version for ARM). TimerShield was implemented in both kernels and compared against the baseline kernels.

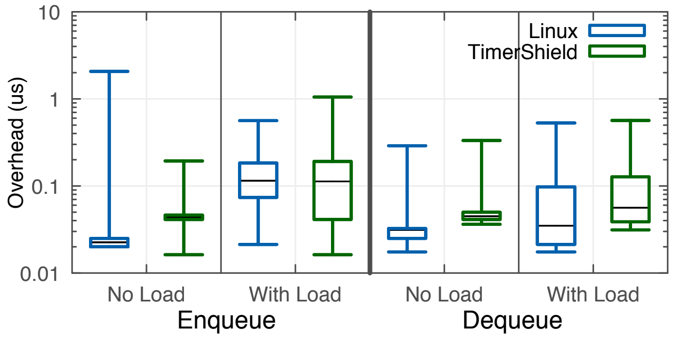

Fig. 2: Timer enqueue and dequeue overheads in μs (Intel).

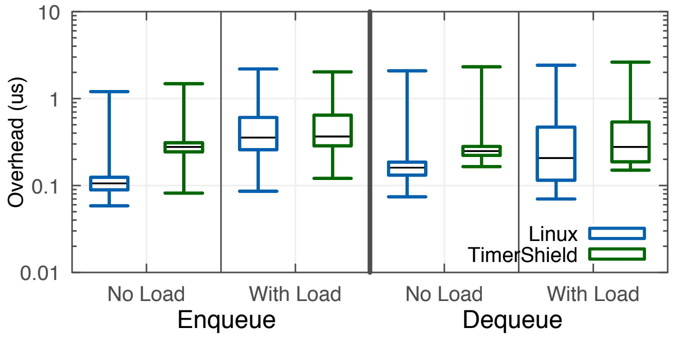

Fig. 3: Timer enqueue and dequeue overheads in μs (ARM).

The task sets and task periods used in the subsequent experiments were modeled around automotive benchmarks proposed by Bosch [22] and use harmonic task periods ranging from 1 ms to 1000 ms. Measurements were carried out by using per-core cycle counters on both the Intel and ARM machines.

B. Overhead Evaluation

We measured *(i)* how much overhead TimerShield adds to timer operations, and *(ii)* how much processing occurs during context switches. We report all overheads in μs (rather than cycles) due to the disparity between the two architectures.

Timer subsystem overhead. Recall that TimerShield uses more complex data structures and thus adds overhead into the timer enqueuing and dequeuing logic. We compared the number of CPU cycles taken to complete the low-level `enqueue_hrtimer()` and `__remove_hrtimer()` functions under both the baseline kernel and TimerShield.

We evaluated two scenarios, one with and one without a background load. Measuring without load evaluated the overhead of an idle system (*i.e.*, with only scheduler timers present). Measuring with load was performed with one high-priority task (with a 1 ms period) and 50 low-priority processes (with harmonic periods ranging from 1 ms to 1000 ms [22]) all using `clock_nanosleep()` to implement periodic activations. This evaluated the overheads associated with a more loaded system that needs to maintain larger data structures.

Fig. 2 and Fig. 3 show the results of our experiments on the Intel and ARM platforms, respectively. Each plot consists of eight "candlestick graphs": the top and bottom edges of the box represent the 5th and 95th percentile overheads, while the top and bottom points of the vertical line represent the minimum and maximum observed overhead. Finally, the horizontal line in the middle of each box represents the median overhead. Note that the Y-axis is plotted in log scale for readability.

Consider for now only the scenarios with no load, where only the scheduler is creating timers that are dequeued on expiry. It can be seen that under TimerShield, both enqueueing and dequeuing (on both platforms) is marginally more expensive: the 95th percentile overhead is increased by 180 ns when enqueuing on ARM under TimerShield compared to Linux, and by 20 ns on the Intel platform. This increase in the cost of timer opertions reflects the overhead of the additional segment tree update under TimerShield, an overhead not incurred under Linux.

However, if we consider the loaded scenario, where many processes are constantly enqueuing timers that are dequeued on expiry, the difference in overhead is much lower. For the 95th percentile, the highest increase in overhead was observed to be for dequeuing with TimerShield requiring an additional 30 ns on Intel and 68 ns on ARM compared to Linux. This lower difference in overheads between Linux and TimerShield overheads in the loaded scenario (compared with the no-load scenario) is because they both incur a comparable overhead in dealing with a single, large rb-tree (since all 50 background processes used the same priority), and this dominates in comparison to the added cost of segment tree updates. In a system with a more varied set of priorities, TimerShield would be dealing with multiple, smaller rb-trees, and would incur even lower overhead.

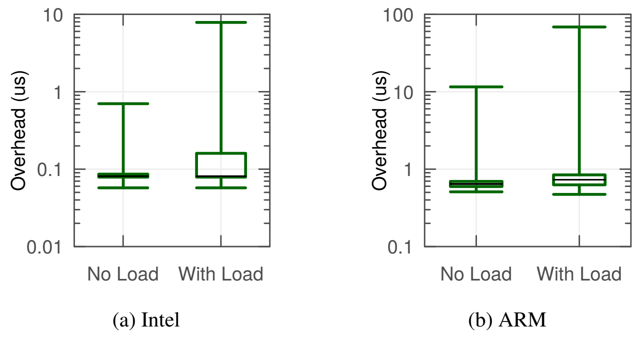

Fig. 4: Additional processing during context switches in TimerShield. Note that the maxima under load include the processing of many deferred timers and do not represent just overhead.

Context-switch overheads. Recall that under TimerShield, we introduce code in the context switch to process expired timers and to reprogram the timer hardware if it resulted in a change in the relevant earliest-expiring timer. We measured these additional costs and results are shown in Fig. 4. Note that, due to limitations in the placement of the tracepoints, Fig. 4 shows both the added overhead *and* the cost of processing deferred timers.

The benchmark setup was similar to the timer subsystem overhead measurements with 50 low-priority processes in the loaded-system scenario. As shown in Fig. 4, the added median cost is under one microsecond on both architectures. Although this is introduced into the scheduler hotpath, the benefits of reduced timer interference (Sec. IV-C) outweigh this overhead.

Code bloat. The amount of code changes performed by Timer-

	Text	Data	BSS
Baseline	10,758	7,402	14,867
TimerShield	10,760	7,437	14,867

TABLE I: Kernel binary segment sizes under baseline Linux and TimerShield (in KiB) on the Intel machine.

Shield to the baseline kernel are small enough so that it can be easily adapted for use in smaller Linux-based embedded systems. Table I compares the size of the text, data, and reserved data segments (BSS) in the baseline kernel binary against those in the TimerShield kernel. As can be seen in Table I, the added code increases the text segment size by approximately 2 KiB.

Memory overhead. Compared to the baseline, TimerShield requires additional memory for two new data structures: *(i)* the priority-aware rb-tree array, and *(ii)* the per-core segment tree.

The size of the priority-aware rb-tree array is determined by the number of priority levels (as there is one rb-tree per priority level). On 64-bit Linux, with 140 priority levels, this amounts to 2,240 bytes per rb-tree array. Similarly, each segment tree, realized as an array of pointers, consists of twice as many nodes as there are priority levels (as a full binary tree with n leaf nodes has a total of $2n - 1$ nodes). Under 64-bit Linux, this also amounts to 2,240 bytes per segment tree.

The total memory overhead in our prototype is actually somewhat higher since we modify Linux's "clock base" abstraction, of which there are several per core (*e.g.*, CLOCK_MONOTONIC, CLOCK_REALTIME, *etc.*), and thus multiple instances of the TimerShield data structures are allocated per core. This can be observed in Table I, which shows a data segment increase of 35 KiB (per core) compared to baseline Linux.

All TimerShield structures are one-time static allocations, incurred just once on boot. Furthermore, the modest increase in memory footprint does not imply a significantly enlarged footprint in the L1 cache, since only a small fraction of each data structure is accessed during a timer operation.

Thus, while not insignificant, we believe this to be an acceptable overhead for Linux-class embedded systems, which typically have multiple megabytes of RAM. Further, as shown next, the relatively small increases in text and data segment sizes enable significantly reduced interference for high-priority tasks.

C. High-Priority Tasks and Interrupt Interference

We evaluated whether TimerShield improves the predictability of high-priority tasks in the presence of lower-priority, timer-creating tasks by measuring the response time of a high-priority task while varying the number of lower-priority tasks.

Our high-priority task consisted of an approximately $200\mu s$-long matrix multiplication operation that was repeated periodically every 1 ms, implemented as a loop with the matrix multiplication followed by `clock_nanosleep()`. This task was set to use the highest SCHED_FIFO real-time priority.

We measured the response time from when the timer expires (and the associated callback is invoked) until the loop completes and the task sleeps again (in `do_nanosleep()`). This takes into account the execution time of a single iteration of the control loop, as well as the overhead of a context switch (which is potentially longer under TimerShield because of the overhead of needing to reprogram the timer to mask lower-priority timers). Timestamps on each machine were recorded via Linux's *ftrace* tracing facility, and 100,000 samples were collected per run.

We evaluated two scenarios: unsynchronized and synchronized task releases of lower-priority tasks. Under unsynchronized releases, the lower-priority timers end up expiring *during* the execution of the high-priority task (as can commonly occur when tasks have sporadic releases or when task sets have non-harmonic periods), thus illustrating the effect of preemptions on the response time of the high-priority task.

In the synchronized case, both low priority and high priority tasks were assigned harmonic periods (with the highest-priority task having the shortest period), and all tasks were released synchronously at a common "time zero". As a result, multiple lower-priority timers expired together *at context-switch boundaries* (and not during the execution of the high-priority task). This results in a cascade of expirations that require immediate servicing, illustrating the importance of priority-aware timer expiration processing as done in TimerShield.

In both experiments, we varied the number of lower-priority tasks (from 1 to 100 on the Intel machine, and from 1 to 80 on the ARM machine, in increments of 1). For clarity, we show results for only three configurations each: a single low-priority task, the maximum number of tasks (100 on Intel and 80 on ARM), and one intermediate result (50 on Intel and 40 on ARM).

Unsynchronized low-priority tasks. We used the popular *cyclictest* [2] program running at a lower SCHED_FIFO real-time priority of 70 for low-priority tasks. We chose cyclictest as it repeatedly sets and expires timers via the `clock_nanosleep()` system call with a configurable period. Periods were chosen uniformly at random from the set of real periods reported in [22] to emulate the range of periods commonly found in practice. Cyclictest instances were launched using a shell script. Since cyclictest determines its intended activation times relative to the start time of the process, this approach results in an unsynchronized workload.

Figs. 5 and 6 show the results of this experiment in the form of a cumulative distribution graph. The X axis shows the response time of the high-priority task (in μs) while the Y axis shows the cumulative probability (*i.e.*, for a given response time, the ratio of samples that were observed to be less than or equal to it). In all following graphs, solid lines represent TimerShield while dashed lines represent the results for baseline Linux.

As can be seen in both figures, while the response-time distribution of the high-priority task is comparable in both TimerShield and baseline Linux when only a single low-priority task creates timers, it degrades significantly under baseline Linux for a larger number of low-priority tasks. This is particularly evident on ARM due to the slower CPU speed and smaller cache size. TimerShield does not experience such an effect on either architecture, regardless of the number of low-priority tasks.

The response time in baseline Linux increases because, as the number of low-priority timers increases, the highest-priority

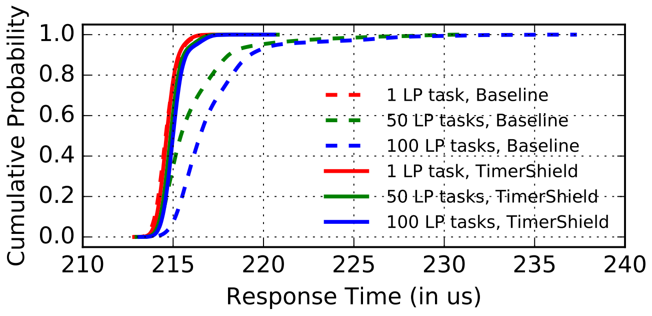

Fig. 5: CDF of high-priority task response times with unsynchronized low-priority background tasks (Intel platform).

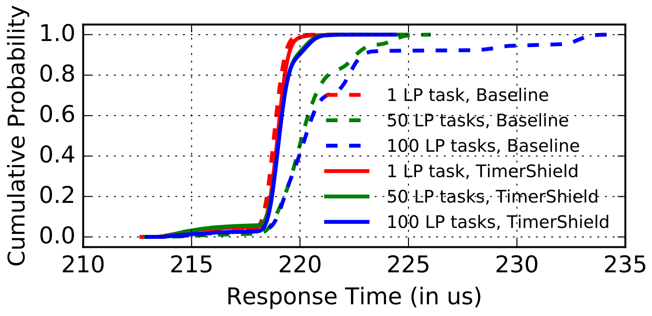

Fig. 7: CDF of high-priority task response times with synchronized low-priority background tasks (Intel platform).

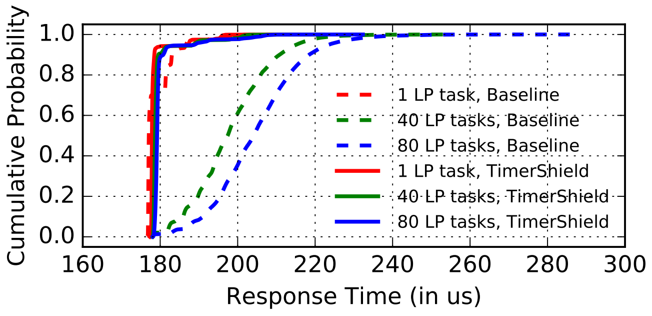

Fig. 6: CDF of high-priority task response times with unsynchronized low-priority background tasks (ARM platform).

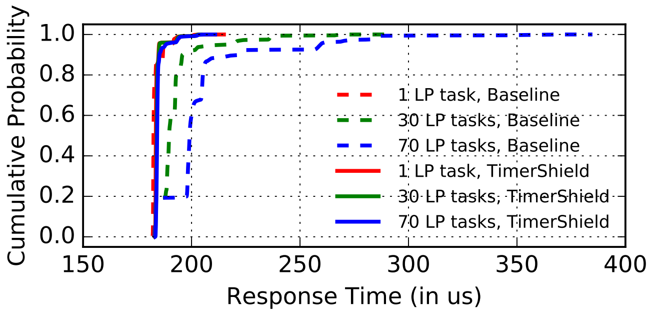

Fig. 8: CDF of high-priority task response times with synchronized low-priority background tasks (ARM platform).

task is preempted more often and thus delayed. In the worst case, this interference is potentially unbounded and restricted only by the number of background tasks creating timers, which is highly unfavorable for real-time guarantees. And even if the number of timers can be bounded *a priori*, the possibility of timer interference introduces unnecessary pessimism into the estimation of worst-case response times.

In contrast, under TimerShield, lower-priority timer interrupts never preempt the high priority task, and are dealt with only once the high priority task completes (and calls `clock_nanosleep`). As a result, we do not see an increase in the response time of the high-priority task. A negligible shift to the right is observable between the curves for the single-task and fifty-task scenarios, and we measured this (using performance counters [32]) to be caused by more frequent cache misses due to the higher number of low-priority processes. However, this affects TimerShield and baseline Linux alike.

Synchronized low-priority tasks. In the synchronized case, we wrote a `pthreads`-based lower-priority task similar to cyclictest but ensured a synchronous release of all tasks by timing wake-ups using a common "time zero." Similar to cyclictest, the task also repeatedly sets and expires timers via the `clock_nanosleep()` system call with a configurable period. Once again, periods were randomly-assigned from between 1 ms and 1000 ms [22]. Figs. 7 and 8 show the CDFs from this experiment, similar to the unsynchronized case.

The figure shows how priority-unaware expiration logic, as it is used in baseline Linux, results in increased response times for high-priority tasks when a cascade of timer expirations occurs. Even though there is no interrupt interference from low-priority task interrupts while the high-priority task executes, the baseline kernel shows increased response times with an increase in background tasks. This occurs because, as there is no notion of timer priority under baseline Linux, all timers are serviced sequentially regardless of priority. As a result, the high-priority task ends up waiting for lower-priority timers to be serviced even though they cannot be delivered immediately.

This situation does not affect the response time of the high-priority task under TimerShield as timer interrupts are processed in a priority-aware manner: the highest-priority timers are picked and serviced one by one, and if at any point this results in a high-priority task waking up, then the process is cut short and the task is scheduled immediately. As a result, there is no wakeup interference before the execution of the high-priority task and the response time does not increase. Once again, we see a slight right-shift in the response-time CDF due to increasing cache misses from the increasing number of low-priority processes (this applies to both TimerShield as well as the baseline kernel).

Note that in both the synchronized and unsynchronized cases, TimerShield performs slightly *worse* with a small number of lower-priority tasks due to its use of more complex data structures. However, this added complexity becomes advantageous quickly: TimerShield begins to outperform baseline Linux with three LP tasks (synchronized case) and five LP tasks (unsynchronized case) under Intel, and three LP tasks under ARM (both synchronized and unsynchronized cases).

Unprivileged low-priority tasks. To further emphasize the problem, we demonstrate how an unprivileged, potentially

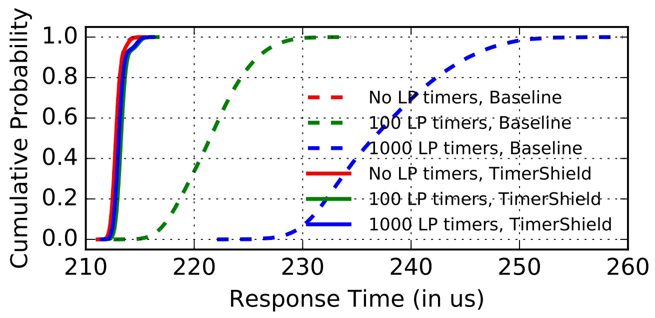

Fig. 9: CDF of high-priority task response times with user-priority (SCHED_NORMAL) timers (Intel platform).

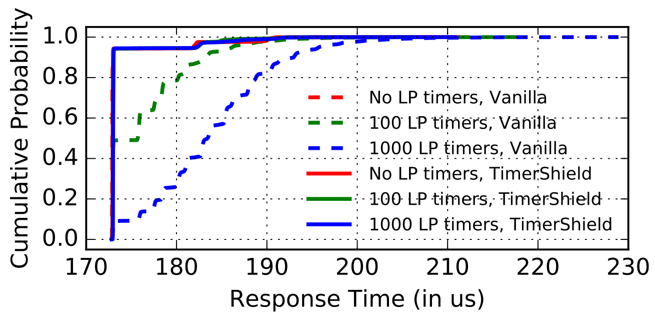

Fig. 10: CDF of high-priority task response times with user-priority (SCHED_NORMAL) timers (ARM platform).

malfunctioning or hostile userspace task with non-real-time priority and without root privileges under baseline Linux can significantly increase the response time of higher-priority tasks as well as lower their throughput. We also demonstrate how TimerShield prevents such unprivileged background tasks from having adverse effects on response times or throughput, a common assumption in real-time scheduling theory.

We simulated a hostile userspace task using Linux's `timerfd` API to create a large number of timers in userspace [20]. The `timerfd` API allows creating timers (with nanosecond granularity, similar to `clock_nanosleep()`) whose expiration notifications can be read from a file descriptor. This allows programs to check for a large number of timer expirations easily using the familiar `select()`, `poll()`, or `epoll()` file-descriptor polling mechanisms. Our primary reasons for using this interface in this example are *(i)* to demonstrate the versatility of TimerShield under different timer interfaces, and *(ii)* because the `timerfd` interface allows a single unprivileged process to manage multiple concurrent timers, unlike `clock_nanosleep()`, with which each process can create only one kernel timer at a time.

We present results from two variants of the hostile task to demonstrate the extent of interference that could potentially be created. The first variant manages 100 periodic timers while the second manages 1,000 timers (note that creating an even larger number of timers is trivially possible). Timers are polled via the `epoll()` system call and their expiration notifications are monitored using a blocking `read()` loop. Timer periods for the first hostile task varied from $1\mu s$ to $100\mu s$ (in increments of $1\mu s$) and that of the second hostile task varied from $1\mu s$ to $1000\mu s$ (again, in increments of $1\mu s$). We employed the same high-priority task as in the previous two experiments.

Figs. 9 and 10 show the results for these experiments for both Intel and ARM platforms. On both platforms, the baseline kernel shows a drastic degradation in the response time of the high priority task in the presence of the hostile userspace task. Thus under a baseline kernel, a high-priority task would need to account for interference from unprivileged, non-real-time userspace tasks that use timers. As the response time increase is proportional to the number of background timers the hostile task creates, the worst-case interference could potentially be very large, and is in fact currently only restricted by the maximum possible number of timers allowed in the operating system.[4] In contrast, the response time distribution of the high-priority task remains entirely unaffected under TimerShield, demonstrating how our approach limits timer interrupt interference from unprivileged userspace tasks.

	Intel		ARM	
	Idle	Hostile	Idle	Hostile
Baseline	7,044.4	6,916.8	471.6	450.6
TimerShield	7,048.3	7,047.0	471.5	471.1

TABLE II: High-priority task throughput in req/ms.

We also measured the throughput of a simple high-priority real-time task that performs as many 3×3 matrix multiplications as possible in a loop. We measured the mean matrix multiplications per millisecond from 10,000 collected samples. Table II compares the mean throughput (for Linux and TimerShield under both Intel and ARM platforms) with and without the 1000-timer variant of the hostile task running in background (denoted "Hostile" and "Idle" in Table II respectively).

As can be seen from Table II, with the introduction of the hostile task in the background, the baseline kernel sees a reduction in mean throughput of ~100 matrix-multiplications per millisecond under Intel, and ~20 matrix-multiplications per millisecond under ARM (*i.e.*, a throughput reduction of 1.8% and 4.4% under Intel and ARM respectively). In contrast, under TimerShield, we observe nearly consistent throughput regardless of whether the hostile background task is running. (The small reduction (~1 req/ms) was again due to increased cache misses within the kernel, from dealing with a large number of timers.)

D. Low-Priority Tasks Under TimerShield

While we have focused on high-priority tasks so far, we also evaluated the effect that TimerShield has on lower-priority processes. In particular, under TimerShield, the *lowest-priority* task stands to suffer the greatest delay as it incurs the data-structure and implementation-related overheads from all higher-priority tasks, as well as the added context-switch overhead.

To evaluate this, we ran tasks of three different priorities. The highest-priority task was run at the highest SCHED_FIFO priority with a period of 1 ms and performed matrix-multiplications (as done in the first two experiments in Sec. IV-C). A varying

[4]In Linux this is only limited by the (configurable) limit on the number of open file descriptors that a process can have.

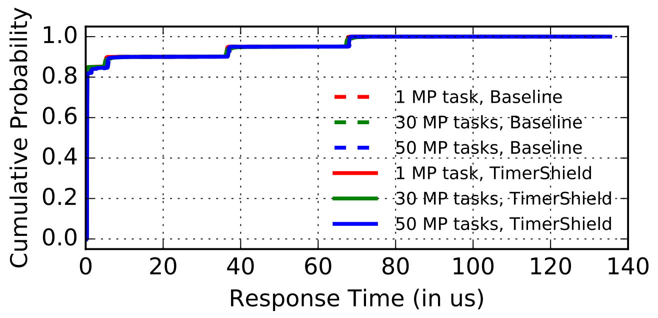

Fig. 11: CDF of response times of lowest-priority task (Intel).

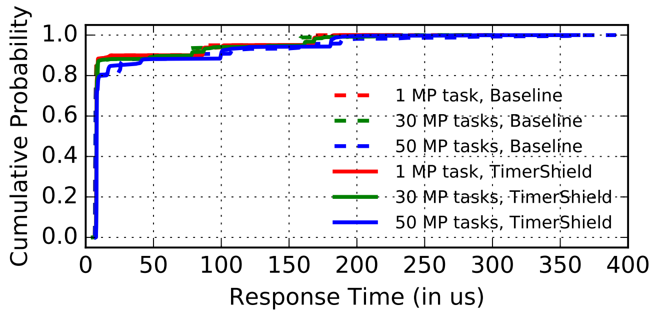

Fig. 12: CDF of response times of lowest-priority task (ARM).

number of medium priority tasks were run at priority 75 under the SCHED_FIFO policy, which additionally interfere with the execution of the low-priority task through periodic sleeps via `clock_nanosleep()`. Finally, we measured the response time of a single low-priority task running at priority 70 under the SCHED_FIFO policy priority with a period of 0.9 ms, which runs a small computation loop per job. We measured the response time of this low-priority task starting from its job release (*i.e.*, when it *would* have started executing in an idle system) to its completion (signaled by the programming of its subsequent wakeup timer). We varied the number of medium priority tasks from 1 to 50; we only show the results for 1, 30, and 50 medium-priority tasks for clarity.

Figs. 11 and 12 show the resulting CDFs of the lowest-priority task on both the Intel and ARM platforms. It can be seen from the figure that there is a negligible increase in the response time of the lower-priority task under TimerShield. This is because the increased overheads of accessing the more-complex data-structures in TimerShield are compensated for by the lowered overhead of servicing expired interrupts in batches rather than one at a time, immediately when they occur. Further, the response time does not increase significantly with an increasing number of medium priority tasks. In fact, in all evaluated configurations, the largest increase in mean response time we observed was $0.06\mu s$ under Intel and $1.79\mu s$ under ARM. We believe this to be an acceptable overhead due to the significantly better predictability of higher-priority tasks.

V. Related Work

High-resolution timer subsystems have been in use for many years. In a Linux context, KURT RTOS [38] was one of the first systems to use one-shot timers to realize high-resolution software timers, which inspired further implementations [13, 23, 33] and related applications [4, 14]. Concerning timer interference, there has been a considerable amount of prior work seeking to account for and/or control interrupts in real-time systems.

Interrupt accounting. A lot of prior work has focused on interrupt accounting for real-time schedulability analysis. A straightforward approach [28] involves treating interrupts as sporadic jobs having the highest priority in the system, and subsequently applying standard fixed-priority response-time analysis. Jeffay and Stone [18] derived an analysis to account for interrupts under *earliest-deadline first* (EDF) scheduling. Sandstrom *et al.* proposed accounting for interrupts by looking at worst-case arrival patterns [37]. More recently, overhead accounting for interrupts has been extended to global scheduling on multiprocessors [7, 9]. TimerShield eliminates a lot of the pessimism involved in accounting for high-resolution timer interrupt overheads as lower-priority timer interrupts can be safely ignored when analyzing high-priority tasks.

Avoiding interrupt interference. The Spring RTOS avoids execution delays due to interrupt handlers entirely by dedicating a separate processor to handling interrupts [39]. Brandenburg and Anderson adapted [8] a similar approach for handling all interrupts under global scheduling, and Cerqueira *et al.* [10] applied the same technique in work on scalable global scheduling. Others have proposed FPGA-based approaches that minimize interrupt interference by incorporating custom interrupt controllers [15, 25, 40]. In contrast, TimerShield is easier to incorporate into existing systems as it is a portable, purely software-based approach. Sáez and Crespo [36] describe an event-driven scheduler that reprograms the hardware timer only when a higher priority task needs to be woken up. This work closely relates to our handling of sleep-based system calls. However, our approach modifies the entire high-resolution subsystem instead of only focusing on scheduler wakeups. Leyva-del-Foyo *et al.* [26] present an approach for integrating the scheduling of IRQs with the kernel scheduler by selectively disabling lower-priority IRQ lines at runtime. This is in spirit similar to what TimerShield does for software timers, but applied at IRQ-level granularity.

Bottom-half scheduling. With split-interrupt handling mechanisms [28], top halves execute at effectively the highest priority (w.r.t. process priorities), but bottom-halves can be executed at any priority. Elliot and Anderson [12] studied the problem of prioritizing bottom halves for GPGPU workloads in PREEMPT_RT. L4-family microkernels [27] (*e.g.*, seL4 [21], Fiasco [1]) integrate bottom halves in an elegant manner: they deliver interrupts to processes as messages via *inter-process communication* (IPC) calls. Jung *et al.* [19] proposed a technique to stabilize execution times of user-space processes by budgeting the CPU time of bottom halves. Scheduling bottom halves by reservation-based mechanisms has also been considered in recent years [29, 30]. QNX also supports budget-sharing and priority-inheritance schemes for interrupt bottom halves [35]. Zhang and West [42] proposed a scheme where the priority of the target process is predicted and the bottom half scheduled with that priority (appropriate enforcement techniques accounted for priority mispredictions). Lee *et al.* [24] improved upon this by focusing on UDP network packet interrupts and thus knowing the priority of the target process *a priori*. Missimer *et al.* [31] proposed the *I/O Adaptive Mixed-Criticality* model with budget enforcement of device bottom halves for use in mixed-criticality systems. However, all of the above approaches still face interference from interrupt top-halves, which for timer interrupts is avoided completely by TimerShield.

VI. Conclusion

We have demonstrated the negative impact of interrupt interference on the predictability of high-priority tasks as it occurs in current designs for high-resolution timer subsystems.

To eliminate such interference, we have proposed TimerShield, a novel high-resolution timer subsystem design that completely avoids lower-priority timer interference while incurring only low overhead. We presented the design and implementation of TimerShield in Linux (TimerShield itself can be adapted to other OSes easily), and compared our prototype with stock Linux PREEMPT_RT on two platforms. Our evaluation shows that the benefits of avoiding interrupt interference justify the minor overhead increase due to the use of more complex data structures. All relevant source code is available online [3].

In future work, it would be interesting to see if TimerShield can be extended to EDF schedulers.

References

[1] "The Fiasco microkernel," https://os.inf.tu-dresden.de/fiasco/.

[2] "Real-time Linux wiki. cyclictest - RTwiki," https://rt.wiki.kernel.org/index.php/Cyclictest, accessed: 2016-10-13.

[3] "TimerShield implementation," available at https://www.mpi-sws.org/~bbb/papers/details/rtas17p, 2016.

[4] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of Linux," in *RTAS*, 2002.

[5] M. Bender and M. Farach-Colton, "The LCA problem revisited," in *LATIN*, 2000.

[6] D. Bovet and M. Cesati, *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[7] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[8] B. Brandenburg and J. Anderson, "On the implementation of global real-time schedulers," in *RTSS*, 2009.

[9] B. Brandenburg, H. Leontyev, and J. Anderson, "An overview of interrupt accounting techniques for multiprocessor real-time systems," *Journal of Systems Architecture: Embedded Software Design*, vol. 57, no. 6, pp. 638–654, 2011.

[10] F. Cerqueira, M. Vanga, and B. B. Brandenburg, "Scaling global scheduling with message passing," in *RTAS*, 2014.

[11] M. Dima and R. Ceterchi, "Efficient range minimum queries using binary indexed trees," *Olympiads in Informatics*, vol. 9, pp. 39–44, 2015.

[12] G. A. Elliott and J. H. Anderson, "The limitations of fixed-priority interrupt handling in PREEMPT_RT and alternative approaches," in *RTLWS*, 2012.

[13] T. Gleixner and D. Niehaus, "Hrtimers and beyond: Transforming the linux time subsystems," in *OLS*, 2006.

[14] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "Supporting time-sensitive applications on a commodity OS," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 165–180, 2002.

[15] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, "Task-aware interrupt controller: Priority space unification in real-time systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 27–30, 2015.

[16] *IA-PC HPET (High Precision Event Timers) Specification*, Intel Corporation, 2004.

[17] *MultiProcessor Specification*, Intel Corporation, 1997.

[18] K. Jeffay and D. Stone, "Accounting for interrupt handling costs in dynamic priority task systems," in *RTSS*, 1993.

[19] K. J. Jung, S. G. Jung, and C. Park, "Stabilizing execution time of user processes by bottom half scheduling in Linux," in *ECRTS*, 2004.

[20] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed. No Starch Press, 2010.

[21] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *SOSP*, 2009.

[22] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmark for free," in *WATERS*, 2015.

[23] J. Lee and K.-H. Park, "Delayed locking technique for improving real-time performance of embedded Linux by prediction of timer interrupt," in *RTAS*, 2005.

[24] M. Lee, J. Lee, A. Shyshkalov, J. Seo, I. Hong, and I. Shin, "On interrupt scheduling based on process priority for predictable real-time behavior," *ACM SIGBED Review*, vol. 7, no. 1, p. 6, 2010.

[25] L. E. Leyva-del Foyo and P. Mejia-Alvarez, "Custom interrupt management for real-time and embedded system kernels," in *RTSS*, 2004.

[26] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Integrated task and interrupt management for real-time systems," *ACM TECS*, vol. 11, no. 2, p. 32, 2012.

[27] J. Liedtke, "On micro-kernel construction," in *SOSP*, 1995.

[28] J. W. S. W. Liu, *Real-Time Systems*, 1st ed. Prentice Hall PTR, 2000.

[29] N. Manica, L. Abeni, and L. Palopoli, "Reservation-based interrupt scheduling," in *RTAS*, 2010.

[30] N. Manica, L. Abeni, L. Palopoli, D. Faggioli, and C. Scordino, "Schedulable device drivers: Implementation and experimental results," *OSPERT*, 2010.

[31] E. Missimer, K. Missimer, and R. West, "Mixed-criticality scheduling with I/O," in *ECRTS*, 2016.

[32] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *DoD HPCMP Users Group Conference*, 1999.

[33] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *RTAS*, 1999.

[34] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction.* Springer, 1985.

[35] *QNX Nutrino RTOS: System Architecture*, QNX Software Systems Limited, 2012.

[36] S. Sáez and A. Crespo, "Integrated schedulers for a predictable interrupt management on real-time kernels," in *Ada-Europe*, 2014.

[37] K. Sandstrom, C. Eriksson, and G. Fohler, "Handling interrupts with static scheduling in an automotive vehicle control system," in *RTCSA*, 1998.

[38] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software," in *RTAS*, 1998.

[39] J. A. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time operating systems," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 3, pp. 54–71, 1989.

[40] J. Strnadel, "Load-adaptive monitor-driven hardware for preventing embedded real-time systems from overloads caused by excessive interrupt rates," in *ARCS*, 2013.

[41] G. Varghese and T. Lauck, "Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility," *SIGOPS Operating Systems Review*, vol. 21, no. 5, 1987.

[42] Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *RTSS*, 2006.