Fast on Average, Predictable in the Worst Case: Exploring Real-Time Futexes in LITMUS$^{\mathrm{RT}}$

Roy Spliet
MPI-SWS

Manohar Vanga
MPI-SWS

Björn B. Brandenburg
MPI-SWS

Sven Dziadek*
TU Dresden

***Abstract*—This paper explores the problem of how to improve the average-case performance of real-time locking protocols, preferably without significantly deteriorating worst-case performance. Motivated by the futex implementation in Linux, where uncontended lock operations under the Priority Inheritance Protocol (PIP) do not incur mode-switching overheads, we extend this concept to more sophisticated protocols; namely the PCP, the MPCP and the FMLP^{+}. We identify the challenges involved in implementing futexes for these protocols and present the design and evaluation of their implementations in LITMUS$^{\mathrm{RT}}$, a real-time extension of the Linux kernel. Our evaluation shows substantial improvements in the uncontended case (*e.g.*, a futex implementation of the PCP lowers lock acquisition and release overheads by up to 75% and 92%, respectively), at the expense of some increases in worst-case overhead on par with Linux's existing futex implementation.**

I. Introduction

Suspension-based real-time locking protocols, such as the classic *Priority Inheritance Protocol* (PIP) [27], are used in real-time systems to enable mutually exclusive access to shared resources while preventing unbounded *priority inversions* [27] (*i.e.*, while preventing lower-priority tasks from blocking higher-priority tasks for potentially unbounded durations). Such blocking, termed *priority-inversion blocking* (henceforth pi-blocking [8]), increases a task's worst-case response time and thus must be fully bounded and accounted for during schedulability analysis.

Practical systems, in addition to accounting for pi-blocking, must also take into account the system overheads associated with lock acquisition and lock release operations. These overheads typically comprise both hardware overheads, such as those incurred by protection mode switches, as well as bookkeeping code in the operating system kernel. Failure to account for these overheads can lead to execution time underestimates and consequently to deadline misses.

Prior work considering locking overheads in real-time systems (reviewed in Sec. VI) has primarily focused on worst-case overheads. However, there are also workloads that can benefit from lower overheads in the common (average) case, *where locks are typically uncontended*. For instance, in soft real-time workloads such as video playback, locks may be acquired many thousands of times per second [6]. Reducing average-case locking overhead in such applications allows for higher throughput (*e.g.*, higher framerates) and fewer deadline misses. Similarly, in mixed-criticality systems, lowering the average-case overheads incurred by high-criticality tasks can aid in accommodating and avoiding deadline misses in lower-criticality tasks. However, while these examples illustrate some of the benefits of low average-case overheads, system schedulability is ultimately determined based on *worst-case overheads*. Thus, a reduction in average-case overheads should ideally not significantly increase worst-case overheads.

An effective approach for achieving these goals is the Linux kernel's support for *fast userspace mutexes* (futexes) [16], a mechanism that supports efficient lock implementations with low average-case overheads. By exporting lock-state information to userspace, futexes avoid expensive system calls when a lock is uncontended, which is arguably the common case in well-designed systems.

While futexes have been successfully applied to implement the PIP in Linux [16], futex-based implementations have not yet been proposed for more demanding protocols such as the *Priority Ceiling Protocol* (PCP) [27], the *Multiprocessor Priority Ceiling Protocol* (MPCP), and the *FIFO Multiprocessor Locking Protocol* (FMLP^{+}) [3, 5]. Given that, on uniprocessors, the PCP offers lower (and in fact optimal) bounds on pi-blocking, and given that, on multiprocessors, specialized protocols such as the MPCP or the FMLP^{+} are required to ensure bounded pi-blocking (as priority inheritance is ineffective across partitions), the lack of futex support for these protocols is a considerable drawback in the context of real-time systems that require both predictability and efficient average-case performance. In this paper, we address this limitation by identifying how futexes may be realized for these three protocols, and by reporting on an empirical assessment of a prototype implementation in LITMUS$^{\mathrm{RT}}$.

The key challenge in extending the PIP futex implementation to more sophisticated protocols reduces to the following difference: the PIP can be considered to be a *reactive* locking protocol, as it takes effect only in reaction to lock acquisition or lock release operations. In contrast, the PCP and multiprocessor locking protocols such as the Multiprocessor Priority Ceiling Protocol (MPCP) and the FMLP^{+} are what we term *anticipatory* locking protocols: they make use of additional information about the workload to anticipate problematic scenarios and prevent them before they can arise. For example, under the PCP, each lock is associated with a priority ceiling that specifies the highest priority of any task that may acquire it, and there exists a system ceiling, which is the lock with the highest priority that is currently being held by some task. If the priority of a task is lower than

*Contributed to this work as part of a bachelor's thesis completed at the University of Saarland, Saarbrücken, Germany.

the system ceiling, it is prohibited from acquiring any locks under the PCP, *even when the lock being requested is currently uncontended*. Similarly, since priority inheritance is ineffective across partition boundaries [9, 24, 25], the MPCP and the FMLP$^+$ employ *priority boosting*, where the priority of jobs in critical sections is boosted past those of all local jobs to guarantee progress. This may result in jobs blocking during lock acquisition even in the absence of resource contention—that is, if implemented literally, kernel interaction is required under these protocols *even in the common case*.

This motivates the key question studied in this paper: how can the futex approach be extended to support anticipatory real-time locking protocols without violating their semantics? Further, can this be done without significantly increasing the worst-case overheads? To this end, this paper makes the following contributions:

- We identify key properties of the PCP, the MPCP, and the FMLP$^+$ (Secs. III and IV) that allow implementing futexes for anticipatory real-time locking protocols.
- We describe futex-based implementations of these protocols (Secs. III and IV) in LITMUS$^{\mathrm{RT}}$ [3, 13]: the PCP-DU, MPCP-DU and FMLP$^+$-DU, respectively. Here, "DU" stands for *deferred update*, as our implementations realize futexes by deferring the update of the state of a lock until the next time the scheduler is called.
- We evaluated these protocols (Sec. V) on an ARM Cortex A9 quad-core system and observed a substantial reduction in uncontended-case overheads (up to 92% under a variant of the PCP-DU), at the expense of an increase in contended-case overheads that is no worse than that experienced by Linux's existing futex implementation.

We begin with some necessary background information.

II. Background and Definitions

We consider a real-time system under *Partitioned Fixed-Priority* (P-FP) scheduling [14], where a set of n *sporadic tasks* [21] is partitioned among m processors in the system, and each processor is scheduled under a fixed-priority scheduler. For a given task T_i, the index i refers to its *base priority* (with T_1 having the highest priority and T_n having the lowest priority). Tasks are scheduled based on their *effective priority*, which may temporarily exceed the base priority under certain resource sharing policies, including each of the policies considered in this paper. For example, the PIP temporarily raises the effective priority of a lock-holding task when a low-priority task blocks a high-priority one. We assume that there is a set of shared resources $R = \{r_1, \ldots, r_n\}$ in the system, each associated with a lock.

Although we assume P-FP scheduling and the sporadic task model throughout this paper, the presented techniques are not limited to P-FP or the sporadic task model and can be easily transferred to other scheduling algorithms and task models.

A. Futexes in Linux

Futexes, short for *fast userspace mutexes*, are a mechanism in the Linux kernel to support the efficient implementation of suspension-based locks in userspace [16]. A basic futex-enabled lock consists of *(i)* a shared integer that contains the current state of the lock, and *(ii)* a kernel-side wait-queue exposed through the futex API. Operations for locking and unlocking futexes are implemented using atomic *fetch-and-increment* and *fetch-and-decrement* instructions, respectively, that operate on the shared lock state.

When attempting to acquire a lock, observing a value of zero indicates that the futex was free and that it has been acquired (the *uncontended case*), while a non-zero value indicates that the futex could not be acquired (the *contended case*, in which case the kernel is invoked in order to block the process).

Similarly, when unlocking a futex, the fetched value indicates either that no action needs to be taken (the uncontended case), or that there are tasks waiting for the futex (the contended case, where the kernel is invoked in order to unblock one of the waiters). The key advantage of futexes is that processes avoid invoking the kernel, via a (relatively) expensive system call, in the uncontended cases.

B. Locking in LITMUS^{RT}

We implemented the protocols presented in this paper in LITMUS$^{\mathrm{RT}}$ [3, 13], a real-time extension of the Linux kernel, as it already includes support for the sporadic task model, P-FP scheduling, and helpful infrastructure for real-time locking protocols. Besides several multiprocessor scheduling policies, LITMUS$^{\mathrm{RT}}$ also implements a variety of locking protocols, including the PCP [27], MPCP [24], the original FMLP [2], and the more recent FMLP$^+$ [3, 5], which have been designed primarily with low analytical pi-blocking bounds and acceptable worst-case runtime overheads in mind.

Under LITMUS$^{\mathrm{RT}}$, real-time tasks use a mutex-based locking API to ensure mutually exclusive access to shared resources. Locks are referenced by userspace tasks using *lock descriptors* [7], which are simply zero-based indices into a per-task lookup table, analogous to UNIX file descriptors. Special system calls provided by LITMUS$^{\mathrm{RT}}$ enable the creation, acquisition, and release of locks associated with a particular protocol. These routines track the state of a lock, suspend tasks when the protocol prevents a particular task from acquiring a lock, and resume tasks once they are eligible to continue. *Importantly, both locking and unlocking require the invocation of system calls even in the uncontended case.*

C. Priority Inheritance Protocol (PIP)

The conceptually simplest of the considered suspension-based real-time locking protocols is the Priority Inheritance Protocol (PIP) proposed by Sha *et al*. [27]. Under the PIP, when a high-priority task blocks on a lock currently held by a lower-priority task, the latter inherits the priority of the high-priority task, thus preventing the possibility of unbounded priority inversion. Linux's PRIO_INHERIT protocol (available as part of the *pthreads* library) is a futex-based implementation of the PIP that does not incur system call overheads for uncontended lock operations.

Although PRIO_INHERIT provides low average-case overheads, the PIP has several properties that make it sub-optimal

for use in certain applications. First, locking under the PIP is susceptible to transitive pi-blocking [27], which occurs when a *chain* of blocked jobs forms so that each job is blocked on a resource currently held by the next job in the (ascending priority-ordered) chain. The chain is resolved in reverse-priority order, resulting in the highest priority task being blocked for the duration of all critical sections in the chain. Second, inconsistently ordered nested lock acquisitions risk deadlock under the PIP. These properties make the PIP, from an analytical point of view, a less attractive choice on uniprocessors. Finally, the PIP is ineffective on multiprocessors under partitioned and clustered scheduling [24, 25, 27].

D. Priority Ceiling Protocol (PCP)

The Priority Ceiling Protocol [27] is a classic *anticipatory* uniprocessor locking protocol which addresses the shortcomings of the PIP (on uniprocessors). Under PCP, jobs experience pi-blocking of at most one outermost critical section [27], which is optimal on uniprocessors, and deadlocks are avoided entirely by the precautionary blocking of jobs in anticipation of possible cyclic dependencies, as summarized next.

In the PCP, each resource r_i has a *priority ceiling* $\Pi(r_i)$, which is the highest priority of any task that may lock r_i. During runtime, the scheduler keeps track of a *system ceiling* (denoted $\hat{\Pi}(t)$), which is the currently held lock (if any) with the highest priority ceiling. A job may acquire a lock only if *(i)* either the job's priority exceeds the system ceiling, or *(ii)* if the requesting job holds the resource that defines the current system ceiling (in this case, we say that the requesting job "*owns*" the system ceiling). If neither of these conditions is satisfied, then the requesting job is blocked and, if the blocked job has a higher effective priority than that of the blocking job, its priority is inherited by the current owner of the system ceiling. When a lock is released, the system ceiling is lowered accordingly. When no locks are currently held (*i.e.*, if the system ceiling is undefined), any task may acquire a lock.

LITMUS$^{\mathrm{RT}}$ implements the *Classic* or *Original* PCP (OPCP), where the system ceiling is maintained explicitly and the priority of a job is raised only when a higher priority job becomes blocked on a lock currently held by the lower-priority job. In particular, the priority of a lock-holding job is raised only to that of the highest-priority job *currently* blocked by it, which minimizes the occurrence of priority inversions.

In contrast, Linux's implementation of the PCP (available under the name PRIO_PROTECT in the *pthreads* library) is an implementation of the *Immediate Priority Ceiling Protocol* (IPCP) [1, 11, 27, 28], which is subtly different from the OPCP. Under the IPCP, when a job acquires a lock, its priority is *immediately* raised to the priority ceiling of the lock, regardless of whether it actually blocks any higher-priority tasks at the time. While this rule may occassionally result in superfluous priority inversions, the IPCP still guarantees an identical bound on worst-case pi-blocking and requires fewer preemptions. Further, explicit maintenance of a system ceiling is not required under the IPCP as it is implicitly represented by the effective priority of lock-holding jobs.

Although PRIO_PROTECT is implemented in Linux using the futex API, it does not enjoy the benefits of the futex approach. The PCP's reliance on priority ceilings (*i.e.*, its anticipatory properties) are not captured in the futex API, which means that explicit system calls are still used to raise or lower a job's priority, even if the lock is uncontended.

E. Multiprocessor Priority Ceiling Protocol (MPCP)

The introduction of parallelism in multiprocessor systems adds a new dimension of complexity to real-time locking protocols, as they now need to account for *remote blocking*, where a job may be blocked by jobs executing concurrently on other processors. Unfortunately, the analytical progress guarantees provided by priority inheritance, which is also a key component of the PCP, break across partition boundaries and do not ensure bounded pi-blocking in all cases [9, 24, 25].

Rajkumar *et al*. proposed the *Multiprocessor Priority Ceiling Protocol* (MPCP) [24], the first shared-memory multiprocessor real-time locking protocol. The MPCP is based on *priority boosting*, where the priority of jobs in critical sections is temporarily raised to a level higher than any used base priority, in order to ensure the progress of lock-holding jobs despite the lack of priority inheritance. Lock-holding jobs cannot be preempted by newly released jobs (which have not entered a critical section yet), but may still be preempted by other priority-boosted jobs.

Under the MPCP, for every partition in the system, each lock is assigned a priority ceiling, which is the highest priority of all tasks in every other partition accessing this lock [24]. When entering a critical section, the priority of a job running in a given partition is boosted to a value relative to the ceiling for this partition. In the case of contention, tasks suspend and gain access in order of decreasing base priority. That is, when a critical section completes and the lock is released, jobs blocked on that lock are resumed in order of priority. Deadlocks are avoided in the MPCP by prohibiting the nesting of locks.

Locking protocols based on priority boosting are anticipatory as they boost the priority of jobs in critical sections to defend against possible future job releases. As untimely future job releases may occur at any time, priority boosting is required even when a lock is initially uncontended, which renders the MPCP incompatible with Linux's futex API for reactive locking protocols.

LITMUS$^{\mathrm{RT}}$ implements the MPCP using the same APIs as the PCP. Upon opening a lock, the priority ceilings for all partitions in the system are set up. When a job tries to acquire the lock, its priority is boosted and an attempt is made to obtain the lock, which is granted immediately if it is uncontended. On contention, however, the job is added to the priority wait-queue associated with the lock and suspended. On releasing the lock, the unlocking job's effective priority is restored to the job's base priority and the next job on the waitqueue (if any) is resumed. As is the case with the PCP, LITMUS$^{\mathrm{RT}}$'s MPCP implementation requires tasks to invoke system calls even in the absence of contention.

F. FIFO Multiprocessor Locking Protocol (FMLP$^+$)

The *FIFO Multiprocessor Locking Protocol* (FMLP$^+$) [3, 5], a refinement of Block *et al.*'s original FMLP [2], is a suspension-based anticipatory locking protocol for partitioned scheduling with an asymptotically optimal bound on maximum suspension-aware pi-blocking ($O(n)$). While the FMLP$^+$ uses priority boosting similar to the MPCP to ensure progress of jobs in critical sections, the key difference is its use of FIFO ordering both for waiting jobs and jobs executing critical sections: under the FMLP$^+$, jobs gain access to contended locks in order of the time at which they issued the lock request, and priority-boosted jobs are also scheduled in order of non-decreasing lock-request times. Similar to the MPCP, nesting of critical sections is forbidden, thus avoiding deadlocks.

The FMLP$^+$ implementation in LITMUS$^{\mathrm{RT}}$ differs slightly from the MPCP implementation: no ceilings need to be determined when opening a lock, the per-lock priority wait-queue is replaced with a FIFO ordered wait-queue, and the priority-boosting mechanism orders boosted jobs based on the time at which they made the lock requests.

With the necessary background in place, we next identify properties of the PCP, the MPCP, and the FMLP$^+$ that allow us to derive futex-like variants of the protocols, and discuss our proof-of-concept implementations of the resulting protocols in LITMUS$^{\mathrm{RT}}$, which indeed avoid involving the kernel in the common case. We chose these protocols because they respectively are the state-of-the-art locking protocols for uniprocessors and partitioned multiprocessors, and because matching non-futex implementations are available as a baseline in LITMUS$^{\mathrm{RT}}$; however, the techniques discussed in the following can be similarly applied to other (multiprocessor) locking protocols as well. We begin with the simpler uniprocessor case in Sec. III and discuss how to apply similar techniques to the MPCP and the FMLP$^+$ in Sec. IV.

III. Uniprocessor Protocols: PCP Futexes

Recall that under the PCP each lock has a priority ceiling, and that the currently held locks define the system ceiling, which in turn determines the outcome of lock acquisition attempts. A global view of the state of all the locks in the system, which is available only to the kernel, is hence required to correctly implement the PCP. The kernel must also be informed of lock acquisitions and releases: acquisition of a lock, if permitted by the current system ceiling, likely raises the system ceiling. Similarly, releasing a lock may potentially lower the system ceiling and hence unblock waiting tasks.

However, in many cases, it is not necessary to inform the kernel *immediately*, or even at all—communicating changes in lock ownership may be safely *deferred* in the common case, often to the point where they become irrelevant so that they may be omitted entirely.

This property may be leveraged to obtain futexes with OPCP semantics, which we argue more clearly in the following by stating the three observations upon which our solution rests. Similar observations were also recently made by Züpke *et al*. [30] in concurrent work exploring IPCP futexes.

Observation 1. *The success or failure of future lock acquisitions by a particular job can be determined immediately before it is dispatched.* Recall that, in the PCP, a job may acquire *any* lock only when its effective priority is greater than the current system ceiling (or if it is the owner of the system ceiling), regardless of the identity of the lock. Since the PCP is a uniprocessor protocol, the system ceiling cannot be changed by other tasks while a job is occupying the processor; the outcome of lock operations may thus be predetermined by the scheduler when dispatching a job. For example, although a higher-priority job may be released at any time, which in turn may acquire a lock and raise the system ceiling, it can do so only after the scheduler has been invoked to preempt the previously executing job.

Observation 2. *The set of jobs blocked on the system ceiling cannot grow while the ceiling owner is executing.* Again, since the PCP is a uniprocessor protocol, newly released jobs can block on the system ceiling only after preempting the previously executing job.

Observation 3. *Updating the state of a lock can be deferred until a context switch occurs.* Similar to Observation 1, any lock contention is preceded by a context switch. Hence, a context switch is the latest time until which we can defer communicating the state of all locks to the kernel.

From Observation 1, it is obvious that it is not necessary to query the kernel's permission for every lock acquisition. From Observation 2, it follows that it is not necessary to inform the kernel of every lock release. And finally, from Observations 1 and 3, we conclude that any ceiling updates can be safely deferred until the kernel is entered anyway to enact a preemption. In particular, if a lock is acquired and released before a preemption occurs—the common case—then the kernel does not have to be aware of the critical section at all. Our implementation exploits this as described next.

A. Implementing Deferred-Update Futexes (PCP-DU)

In this paper, we present futexes with *classic* PCP semantics (*i.e.*, we implement the OPCP, as opposed to the IPCP). The advantages of the classic PCP are that it does not require userspace processes to be aware of the ceilings of locks, which simplifies system integration, and that it avoids superfluous priority inversions (that may arise under the IPCP due to the unconditional elevation of a lock holder's effective priority).

The central data structure underlying our futex implementations, as shown in Listing 1, is a bidirectional communication channel called a *lock page* (lines 3–6 in Listing 1), a per-task shared page of memory mapped into the task's address space, which both the task and the kernel can read and modify.

To realize OPCP futexes, we must provide two mechanisms: *(i)* a mechanism that allows userspace processes to let the kernel know which locks were acquired or released between two scheduler invocations (Observation 3), and *(ii)* a mechanism to implement ceiling blocking. We now describe how both these mechanisms are implemented using the lock page structure.

Which locks were acquired? To asynchronously inform the kernel of lock acquisitions and releases, the lock page contains a bitmap named `locked` (line 4), which tracks for each lock

```
1  #define SIZE divide_ceiling(MAX_LOCKS, BITS_PER_WORD)
2
3  struct lock_page {
4      bitmap_t locked[SIZE];
5      bool unlock_syscall;
6  } *lock_page;
7
8  void lock(int desc)
9  {
10     lock_page->locked[IDX(desc)] |= (1 << BIT(desc));
11 }
12
13 void unlock(int desc)
14 {
15     lock_page->locked[IDX(desc)] &= ~(1 << BIT(desc));
16     if (lock_page->unlock_syscall)
17         kernel_do_unlock(desc);
18 }
```

Listing 1: PCP-DU-PF userspace routines. The `IDX()` and `BIT()` functions return, for a given lock descriptor, the index of the entry and the corresponding bit in the bitmap, respectively.

```
1  #define CAN_LOCK (1 << (BITS_PER_WORD - 1))
2  ...
3  void lock_bool(int desc)
4  {
5      old = lock_page->locked[IDX(desc)] | CAN_LOCK;
6      new = old | (1 << desc);
7      if (!CMPXCHG(&lock_page->locked[IDX(desc)], old, new))
8          kernel_do_lock(desc);
9  }
```

Listing 2: The PCP-DU-BOOL locking operation. The `CMPXCHG` operation takes an address, an expected value, and a new value, and atomically stores the new value at the address if the currently stored value equals the expected value. The `IDX()` and `BIT()` functions have been redefined to account for the `CAN_LOCK` flag.

descriptor whether the task currently holds the corresponding lock. In accordance with Observation 3, if the lock is released (and the corresponding bit cleared) before the scheduler is invoked, the kernel never takes note of a critical section.

How is ceiling blocking ensured? Based on Observation 1, when a task is dispatched, the kernel must communicate whether the task may acquire any locks. We implemented this mechanism in two different ways: the first method, named PCP-DU-PF, exploits the exception handling mechanism provided by the *memory management unit* (MMU), while the second approach, named PCP-DU-BOOL, uses the atomic *compare-and-exchange* operation (henceforth `CMPXCHG`) to decide whether the kernel should be invoked.

B. PCP-DU-PF: Ceiling Blocking with Page Faults

Under the PCP-DU-PF, the kernel uses the MMU to be automatically notified of failed lock attempts by means of a page fault. To this end, the kernel remaps the lock page as a read-only page if the task is not allowed to acquire locks. Lock acquisition thus consists simply of attempting to write a bit to the `locked` bitmap array (see line 11 in Listing 1). The lock is acquired when this write operation succeeds, otherwise an access violation fault is triggered, which allows the kernel to block the requesting process while letting the current ceiling owner inherit the blocked task's priority.

When releasing a lock, two cases are possible: if no other tasks are blocked on the system ceiling, then simply clearing the corresponding bit in `locked` is sufficient (line 15). Otherwise, the kernel should be invoked to resume all tasks with priorities now exceeding the new system ceiling (if any). Based on Observation 2, the kernel communicates whether there are any existing waiters before scheduling a job using the `unlock_syscall` flag in the lock page (line 5).

Note that checking the `unlock_syscall` flag (line 16) and invoking the kernel (line 17) is not atomic. In the rare case of a change in the `unlock_syscall` field due to a preemption occurring between the two steps, the kernel may unnecessarily be invoked, which causes some avoidable overhead, but does not result in incorrect behavior.

As a final corner case, the lock release operation may find the lock page to be unwritable, which can occur if a higher-priority job is released, acquires a lock (raising the system ceiling), and then self-suspends. In this case, a page fault is triggered in line 15 and the releasing job is simply blocked until the ceiling is lowered, after which the unlock operation is completed, which is compliant with OPCP semantics.

In our implementation, the lock page does not reveal *which* acquired lock(s) define the current system ceiling. In the case of contended *nested* locks, it may happen that a nested lock release does not lead to a lowering of the system ceiling, which results in a superfluous system call. However, accounting for this corner case defeats the primary goal of futexes: the more complex logic required would increase both the average- and worst-case locking overheads in return for slightly improved efficiency in an uncommon case.

The key advantage of PCP-DU-PF is that there are no atomic operations or branches in the (frequently executed) lock acquisition code (lines 8–11), which only unconditionally updates a single word of memory, at the expense of incurring a page fault in the (relatively rare) case of ceiling blocking.

C. PCP-DU-BOOL: Checking the Ceiling with `CMPXCHG`

Our second approach, named PCP-DU-BOOL, does not use the MMU, but instead relies on explicitly setting a flag in a task's lock page prior to dispatching it to indicate whether locks may be acquired. However, the checking of this flag and the acquisition of a lock must now be carried out atomically, since a preemption between the two steps could otherwise result in a *time-of-check-to-time-of-use* race condition.

In our implementation, the required atomicity is realized with an atomic `CMPXCHG` operation, as shown in Listing 2. Since in most conventional architectures `CMPXCHG` operates only on individual words, the format of the `locked` array is slightly changed: a single bit, named `CAN_LOCK` bit, is reserved in every word comprising the `locked` bitmap. As implied by the name, the `CAN_LOCK` flag specifies whether locks may be acquired or not, and it is replicated across the `locked` array so that it may be queried within a `CMPXCHG` operation when updating any bit. This is apparent in lines 5–7 of Listing 2, which together ensure that the `CMPXCHG` operation on line 7 fails if the `CAN_LOCK` flag is unset.

The unlocking code remains conceptually unchanged. The advantage of the PCP-DU-BOOL variant is that it avoids page faults, at the expense of requiring an atomic operation as part of every lock acquisition operation, and an additional branch when the lock is contended.

IV. Multiprocessor Protocols: MPCP and FMLP$^+$

Supporting the MPCP and the FMLP$^+$ may appear to be slightly more difficult due to *(i)* remote blocking, which prevents us from deferring the update of a lock's state until the next context switch, *(ii)* priority boosting, which should take effect even if no contention is initially encountered, and *(iii)* since, in the case of the FMLP$^+$, boosting depends on the time at which a request is issued (which seemingly indicates that the kernel must always be notified of lock acquisitions). In fact, there is a simple solution for each of these issues.

Concerning (i), lock acquisitions must be immediately globally visible since locks may be requested concurrently from remote processors. We thus simply adopt Linux's approach and use atomic operations to let tasks acquire and release locks in userpace, which implies that each lock's state must be stored in shared memory visible to all tasks sharing a lock.

Concerning (ii), we observe that whether a task executes with an elevated priority is relevant only to the *local* scheduler, which implies that the priority boosting of critical sections can be deferred until the next context switch, just as it is the case with system ceiling updates in the PCP-DU variants.

Finally, concerning (iii), it may seem that the kernel must know of every lock acquisition under the FMLP$^+$, or that tasks at least need to provide a suitably accurate timestamp in the lock page. Fortunately, the recording of the timestamp can actually be safely deferred until the next context switch, too.

We discuss this observation in more detail in Sec. IV-B, after first introducing the simpler MPCP-DU, which realizes futexes with MPCP semantics.

A. The MPCP with Deferred Updates (MPCP-DU)

Listing 3 shows the pseudocode for MPCP-DU. Similar to the PCP-DU variants, the multiprocessor implementations use a shared lock page. We must provide two key mechanisms to implement futexes for the MPCP: *(i)* the scheduler needs to detect whether the currently executing task is priority boosted, and *(ii)* the executing task needs a way to detect when remote jobs are blocked on a lock.

Deferring priority boosting. Similar to the PCP-DU variants, we use a bitmap in the lock page (called `boost`, line 4 in Listing 3) that specifies whether a particular lock is currently being held by the task (updated in lines 10 and 22). We chose to use a bitmap for this purpose since the MPCP-DU and FMLP$^+$-DU implementations are layered on top of the common PCP-DU code in our prototype. However, since nested lock acquisitions are disallowed under both the MPCP and FMLP$^+$, one could also just record the descriptor of the currently held lock in an integer field in the lock page.

Note that it is insufficient under the MPCP to simply communicate *that* the current task is priority-boosted; rather, it is also required to communicate *which* lock it currently holds so that the scheduler may infer the correct ceiling priority.

```
1  #define SIZE divide_ceiling(MAX_LOCKS, BITS_PER_WORD)
2  int counter[MAX_LOCKS]; /* shared among all tasks */
3  struct lock_page {
4      bitmap boost[SIZE];
5      bool unlock_syscall;
6  } *lock_page;           /* private per-task state */
7
8  void lock(int desc)
9  {
10     lock_page->boost[IDX(desc)] |= (1 << BIT(desc));
11     old = atomic_inc(&counter[desc]);
12     if (old > 0)
13         kernel_do_lock(desc);
14 }
15
16 void unlock(int desc)
17 {
18     old = atomic_dec(&counter[desc]);
19     if (old != 1) {
20         kernel_do_unlock(desc);
21     } else {
22         lock_page->boost[IDX(desc)] &= ~(1 << BIT(desc));
23         if (lock_page->unlock_syscall)
24             sched_yield();
25     }
26 }
```

Listing 3: MPCP-DU/FMLP$^+$-DU userspace routines. The `IDX()` and `BIT()` functions are defined as in Listing 1.

An alternative design would be to explicitly set the current priority (*e.g.*, see [30]); however, this requires each task to be aware of the priority ceilings of all resources that it accesses, which complicates system integration. In LITMUS$^{\mathrm{RT}}$, the ceiling is instead dynamically determined by the kernel [7].

Tracking remote blocking. A shared atomic counter is used to signal that remote jobs are blocked on a given lock. In Listing 3, these counters are shown as a shared array of lock states (line 2); however, it is equally possible to embed each counter in other shared data structures. Atomic increment and decrement operations (which return the previously stored value) are used to count the number of (remote and local) tasks currently contending for a lock (lines 11 and 18). This enables tasks to invoke the kernel only when the lock is contended (lines 12 and 19), just as it is the case with Linux's existing futex API (recall Sec. II-A).

However, even when no remote blockers are present, there may still be local tasks with a higher effective priority waiting to preempt the task under the MPCP (this case is not covered by Linux's futex API for reactive locking protocols). In the MPCP-DU, the presence of waiting higher-priority local tasks is indicated using the `unlock_syscall` flag in the lock page, which is checked each time a task unlocks a resource (lines 23–24), just as in the PCP-DU variants.

B. The FMLP$^+$ with Deferred Updates (FMLP$^+$-DU)

Our implemention of futexes with FMLP$^+$ semantics, denoted FMLP$^+$-DU, resembles in large parts the MPCP-DU, but needs to address two additional exceptions.

Ensuring request-time ordering. Recall from Sec. II-F that, under the FMLP$^+$, priority-boosted processes are ordered

based on the time at which they requested the currently held lock (and not the time of acquisition, as in the original FMLP [2, 7] for partitioned schedulers). In the non-futex baseline implementation of the FMLP$^+$ in LITMUS$^{\mathrm{RT}}$, the required timestamp is simply obtained from the high-resolution clock that is also used for scheduling. However, this poses a challenge in the context of the FMLP$^+$-DU: the kernel's accurate clock is unavailable to userspace processes, and invoking the kernel to acquire a current timestamp is fundamentally in conflict with the futex philosophy. Instead, our solution is based on the the following observation.

Observation 4. *The relative order of local timestamps remains unchanged if reading the clock is deferred until the next context switch.* Since a task's effective priority is relevant only to the local scheduler, and since other local tasks cannot issue requests while a job is scheduled, it is sufficient to obtain a timestamp only when the scheduler is invoked anyway.

Therefore, just like in the MPCP-DU, the effective priority of a task is not determined until it is actually interrupted by the scheduler. This enables us to realize futexes with FMLP$^+$ semantics, and further has the benefit that in the common case, in which a job acquires and releases a lock without the kernel taking note of it, the clock readout is omitted entirely.

However, there exists one corner case, as discussed next.

Dealing with repeated acquisitions. Unknown to the kernel, a lock may be released and re-acquired multiple times while a job is scheduled. In particular, if a previously preempted job already holds a lock when it is dispatched, and if that lock is also held by the job the next time it is interrupted by the scheduler, then the kernel cannot infer from the simple `boost` bitmap in the lock page whether the lock was held continuously or whether it was released and reacquired. However, it is crucial to distinguish between the two cases in order to determine the appropriate effective priority (*i.e.*, to check if a new lock-request timestamp should be associated with the task).

As a solution, under the FMLP$^+$-DU, an `unlocked` flag can be added to each lock page, which is set by the task each time after it releases a lock, indicating to the scheduler that at least one unlock operation occurred. The flag is cleared by the scheduler each time it dispatches a task. As a result, under the assumption that locks are not nested (as defined in the FMLP$^+$ specification), the kernel can trivially detect that a lock was released and reacquired from the fact that the `unlocked` flag was set by the task.

C. Further Optimizations

In addition to the presented general techniques, we have identified two system-specific optimizations in the MPCP and the FMLP$^+$ implementations in LITMUS$^{\mathrm{RT}}$ that improve performance in both the average and the worst case. As these improvements are applicable to both the futex and the baseline non-futex implementations, we discuss them separately.

Avoiding transitive spin delays. In the implementations of the FMLP$^+$, MPCP, FMLP$^+$-DU, and the MPCP-DU, the state of each per-lock wait-queue is protected with a spinlock. In the baseline implementations, in the contented case, this spinlock is held during unlock operations while (one of) the blocked task(s) is resumed with Linux's `wake_up_task()` function. However, we have observed that `wake_up_task()` can take several thousand cycles to complete (likely due to contention for remote runqueue locks). By moving the call to this routine outside of the wait-queue critical section, transitive spin delays via the wait-queue locks are avoided, which reduces the overheads of contended lock acquisitions.

Avoiding the scheduler clock. In the baseline FMLP$^+$ implementation, a task's effective priority is determined with the scheduler's high-resolution clock. However, depending on the underlying hardware platform, accessing this clock may incur non-trivial overheads on the order of dozens to hundreds of cycles (*e.g.*, when accessing off-chip clock devices).

Recall from Observation 4 that only the *relative* order of local timestamps is relevant under the FMLP$^+$. We can thus replace the clock-based timestamp with a per-processor counter (*i.e.*, a "logical clock") that is incremented each time a timestamp is taken. This eliminates a large part of the overhead associated with priority-boosting tasks under the FMLP$^+$.

V. Experiments and Results

We evaluated the four proposed futex implementations (PCP-DU-PF, PCP-DU-BOOL, MPCP-DU, and FMLP$^+$-DU) by comparing them against the two standard Linux protocols PRIO_INHERIT and PRIO_PROTECT, the existing LITMUS$^{\mathrm{RT}}$ implementations of the PCP, the MPCP, and the FMLP$^+$, and also against two new implementations of the MPCP and FMLP$^+$, denoted MPCP-NEW and FMLP$^+$-NEW, that include the optimizations proposed in Sec. IV-C.

All experiments were conducted in LITMUS$^{\mathrm{RT}}$ 2013.1 (based on Linux 3.10.5) running on a Boundary Devices Sabre Lite ARMv7 development board, which is based on the FreeScale I.MX6Q SoC, an ARM Cortex-A9 quad-core system running at 1GHz. Several bugfixes and ARM-specific spinlock performance improvements were backported into the 2013.1 tree from LITMUS$^{\mathrm{RT}}$ 2014.1 and Linux 3.13, respectively. The kernel was compiled to the Thumb-2 instruction set and all kernel debugging options were disabled.

In our experiments, we investigated the following questions: *(i)* How do the overheads of our implementations compare with those of the original protocols in the uncontended case? *(ii)* What additional overheads are introduced by the futex-based approach and does it increase the worst-case overheads? *(iii)* Does the PCP-DU-PF tradeoff—avoiding atomic operations in the uncontended case at the expense of page faults in the contended case—pay off in Linux?

To answer these questions, we implemented microbenchmarks that measure lock and unlock overheads with the processor's cycle counter, as discussed in the following.

A. Microbenchmarking Methodology

Our test driver spawns several real-time threads, each of which lock and unlock a single shared resource once every period. Threads were randomly assigned a critical section length in the range of 25–45μs, an execution time in the range of 25–65μs, and a period in the range of 600–800μs. These

	PRIO_INH	PRIO_PROT	PCP	PCP-DU-PF	PCP-DU-BOOL	MPCP-ORIG	MPCP-NEW	MPCP-DU	FMLP+-ORIG	FMLP+-NEW	FMLP+-DU
Type	Samples: 6,600,000 per protocol				**Uncontended Case**			Samples: 12,200,000 per protocol			
Lock (Avg)	263 (-59%)	1,604 (+149%)	645	160 (-75%)	171 (-73%)	1,075 (-1%)	1,091	214 (-80%)	1,363 (+31%)	1,041	215 (-79%)
Lock (99%)	464 (-65%)	2,236 (+70%)	1,313	661 (-50%)	453 (-65%)	1,928 (-3%)	1,992	591 (-70%)	2,209 (+15%)	1,921	653 (-66%)
Lock (Max)	7,707 (+100%)	33,271 (+764%)	3,850	2,384 (-38%)	1,974 (-49%)	5,875 (-10%)	6,506	2,477 (-62%)	7,491 (-5%)	7,906	2,625 (-67%)
Unlock(Avg)	301 (-75%)	1,221 (+1%)	1,211	87 (-93%)	92 (-92%)	1,216 (+6%)	1,149	177 (-85%)	1,181 (+6%)	1,117	174 (-84%)
Unlock(99%)	495 (-74%)	2,001 (+4%)	1,927	197 (-90%)	204 (-89%)	1,757 (+3%)	1,709	285 (-83%)	1,716 (+7%)	1,597	279 (-83%)
Unlock(Max)	7,786 (+17%)	9,027 (+36%)	6,655	1,770 (-73%)	1,653 (-75%)	7,330 (+19%)	6,158	1,920 (-69%)	6,574 (+24%)	5,292	1,875 (-65%)
	Samples: 900,000 per protocol				**Contended Case**			Samples: 2,100,000 per protocol			
Lock (Avg)	4,955 (+152%)	—	1,967	4,625 (+135%)	2,673 (+36%)	1,266 (+4%)	1,216	1,097 (-10%)	1,441 (+31%)	1,096	1,059 (-3%)
Lock (99%)	6,392 (+128%)	—	2,803	6,317 (+125%)	3,818 (+36%)	2,356 (+7%)	2,201	2,120 (-4%)	2,504 (+20%)	2,080	2,064 (-1%)
Lock (Max)	13,252 (+67%)	—	7,953	14,820 (+86%)	10,880 (+37%)	12,242 (+95%)	6,281	5,392 (-14%)	13,109 (+118%)	6,026	5,570 (-8%)
Unlock(Avg)	3,764 (-8%)	—	4,109	5,148 (+25%)	4,868 (+18%)	4,301 (+0%)	4,280	2,657 (-38%)	4,203 (+1%)	4,166	2,335 (-44%)
Unlock(99%)	6,784 (+16%)	—	5,867	7,213 (+23%)	6,889 (+17%)	7,810 (+0%)	7,784	7,184 (-8%)	8,061 (+5%)	7,680	6,524 (-15%)
Unlock(Max)	15,772 (+6%)	—	14,919	15,985 (+7%)	15,787 (+6%)	20,482 (-5%)	21,657	18,258 (-16%)	21,223 (-5%)	22,451	17,991 (-20%)

Table 1: Observed average, 99$^{\text{th}}$ percentile, and maximum overheads for lock acquisition and release operations under each of the considered protocols. The table reports both absolute values (in processor cycles) and change relative to a baseline (the PCP for uniprocessor protocols, the FMLP^{+}-NEW for multiprocessor protocols). Explicit contention does not arise under PRIO_PROTECT in our experimental setup.

parameters were chosen to simulate a demanding workload with a large number of lock acquisitions per second (*i.e.*, the type of workload that likely benefits from the futex approach).

Five threads were spawned for the uniprocessor protocols, while for the multiprocessor protocols, six threads were spawned across three cores. (One core was reserved for tracing and control tasks.) A cache-polluting loop was executed during and around the critical sections to generate cache pressure.

Lock/unlock samples were obtained by reading the processor's cycle counter before and after each operation in userspace, and also when resuming and suspending tasks in the kernel, which allowed in a post-processing step to exclude the time the task was suspended (if at all). Additionally, since our futex protocols require additional checks each time a task is dispatched, we also recorded scheduling overheads using LITMUS$^{\text{RT}}$'s standard overhead tracing facilities [3]. In each run, samples were collected for an interval of five seconds. We conducted 300 runs for each of the eleven protocols, for a total of 275 minutes of traced execution.

To allow an unbiased comparison of the observed maxima, we normalized the sample sizes within each class of protocols (uniprocessor and multiprocesssor) and with respect to each operation by randomly discarding samples from the data sets. The observed average, 99$^{\text{th}}$ percentile, and maximum overheads, along with the final data set sizes, are listed in Table 1. The percentages in parentheses denote the relative change in overhead in comparison to the baseline implementation. The baseline for both uniprocessor implementations (PCP-DU-PF and PCP-DU-BOOL) is the PCP, while the MPCP-NEW and the FMLP^{+}-NEW are the baselines for the MPCP-DU and the FMLP^{+} respectively. (MPCP-ORIG and FMLP^{+}-ORIG denote the existing, unoptimized LITMUS$^{\text{RT}}$ implementations.)

We first consider the overheads in the uncontended case.

B. Uncontended-Case Overhead Reduction

As expected, the futex approach results in significant overhead reductions, which is apparent for each of the proposed protocols. As can be seen in Table 1, in the uncontended case, the overhead of the futex-based implementations compared to the respective baselines shows substantial overhead reductions in the range of 73%–93% in the average case, still in the range of 50%–90% in terms 99$^{\text{th}}$ percentile overheads, and 38%–75% in terms of maximum observed overheads.

For example, both the PCP-DU-PF and the PCP-DU-BOOL require only around 90 cycles for an uncontended unlock operation on average, whereas the original PCP implementation requires about 1,211 cycles on average—a more than 13x reduction in overheads. In general, the observed large reduction in uncontended-case overheads is a direct consequence of the futex approach and thus validates our designs.

Interestingly, both PCP-DU variants exhibit lower average and maximum overheads than Linux's futex implementation (PRIO_INHERIT), and LITMUS$^{\text{RT}}$'s PCP implementation exhibits lower overheads than Linux's non-futex IPCP implementation (PRIO_PROTECT), which suggests that the prototype implementations in LITMUS$^{\text{RT}}$ are reasonably efficient.

C. Contended-Case Overhead Penalty

Next, we consider the increase in contended-case overheads, which is a result of the additional checks and scheduling logic required in our futex protocols. Interestingly, different trends manifested in the uni- and multiprocessor cases.

In the uniprocessor case, both PCP-DU variants exhibit a noticeable increase in contended-case lock acquisition and release overheads. The extreme case is the PCP-DU-PF, which exhibits a 135% increase in lock acquisition overheads in the average case, and still an 86% increase in terms of the observed maximum overheads (an increase to 14820 cycles from 7953 cycles under the PCP). We attribute this large increase to the complex logic and locking scheme in Linux's page-fault handler, which is not optimized for our futex implementation. A more lightweight kernel might, however, find the PCP-DU-PF more suitable given the large overhead reduction in the uncontended case.

The PCP-DU-BOOL exhibits a more modest increase in contended-case overheads of 36%–37% for lock acquisitions, and 6%–18% for lock releases. While this increase is larger than we expected, the PCP-DU-BOOL's overheads are nevertheless still comparable to those of Linux's PRIO_INHERIT implementation. (No overheads are reported for the contended

case under the PRIO_PROTECT because no explicit contention arises under the IPCP unless lock-holding tasks self-suspend.) The large unconteded-case overhead reduction can thus still be expected to be beneficial to workloads that can tolerate the increase in contended-case overheads.

The multiprocessor case paints a more positive picture, but also one that is more suprising: the contended-case overheads *decreased* under the MPCP-DU and FMLP$^+$-DU compared to their respective baselines. This is a positive outcome, as it shows that there is at least no undesirable increase in overheads. However, from an inspection of the code, there is also no apparent reason for why the overheads might have decreased at all. In fact, from first principles, we expected a largely unchanged overhead distribution.

After a careful analysis and additional tracing, we concluded that the apparent overhead reduction is simply a consequence, or rather an artifact, of the measurement-based methodology: as the introduction of the futex fast-path reduces the contention for the in-kernel wait-queue spinlocks, observing samples close to the worst case becomes less likely *given an equal number of samples*. That is, even in the contended case, the futex approach inherently biases the measurements away from the extremum. Unfortunately, this is an inherent limitation of a measurement-based evaluation; it would thus be worthwhile to reassess the costs of our futex protocols in the context of systems in which true worst-case overheads are determined using static worst-case execution time analysis.

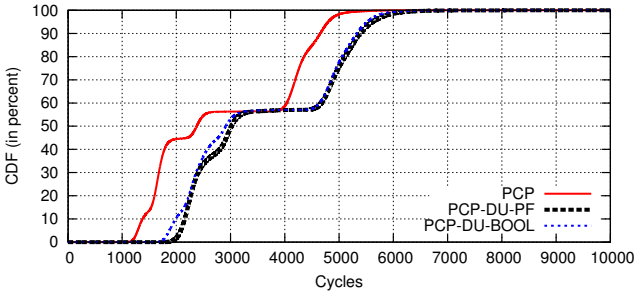
Nonetheless, our results substantiate the benefits of applying the futex approach to multiprocessor real-time locking protocols in Linux and other complex Linux-like systems, for which no static timing analysis is currently available.

D. Scheduler Overheads

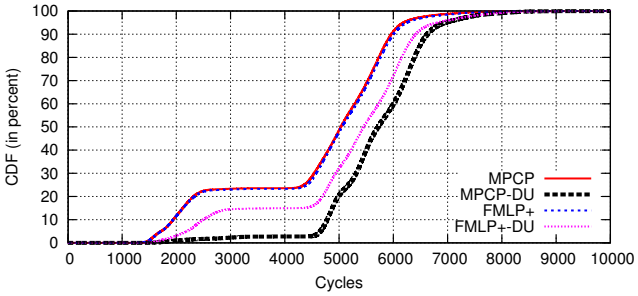
Finally, due to the added checks required each time that a task is dispatched, a possible increase in scheduling overhead must also be taken into account. Fig. 1 depicts the cumulative distribution of the observed scheduling overhead samples under each of the LITMUS$^{\mathrm{RT}}$-based protocols (16,000,000 samples per protocol). Fig. 1(a) depicts the uniprocessor case, which shows that the scheduling overhead is similar under the PCP-DU-BOOL and the PCP-DU-PF, even though the latter requires flushing the lock page's TLB entry when the locking permission changes. Compared to the baseline, both PCP-DU variants incur a fixed additional overhead of around 700 cycles, as indicated by the identical, but shifted shape of all curves.

Fig. 1(b) shows the multiprocessor case, where the MPCP-DU incurs generally more overhead in the scheduler than the FMLP$^+$-DU. This is because the MPCP-DU inserts tasks into a priority queue, which is more costly than appending a task to a FIFO queue, and also since the MPCP-DU must lookup the proper priority ceiling in order to calculate the boosted priority, whereas FMLP$^+$-DU simply uses a logical timestamp. Overall, compared to the baselines, the FMLP$^+$-DU and the MPCP-DU incur a scheduling overhead of approximately 400 and 650 cycles, respectively.

Notably, in both the uni- and the multiprocessor cases, the added scheduling overhead is considerably less than the



(a) Uniprocessor protocols. The curves of the two PCP-DU variants overlap.



(b) Multiprocessor protocols. The MPCP and the FMLP$^+$ curves overlap.

Fig. 1. Cumulative distributions of the observed scheduling overhead

number of cycles saved on a single uncontended critical section, which shows that the futex approach does indeed yield net savings for locking-intensive, low-contention workloads.

VI. Related Work

The priority-inversion problem in real-time systems has been studied extensively, primarily from an algorithmic point of view, on both uniprocessors [1, 27] and multiprocessors (*e.g.*, [2, 3, 5, 8, 12, 18, 20, 23, 24]); see [3, 5, 12] for recent surveys. While several prior studies have considered overheads in suspension-based locking protocols [3, 4, 10, 17, 22], these studies have considered non-futex implementations [3, 4, 10] or used systems without a kernel/userspace separation [17, 22], where mode-switching overheads do not arise.

Mode-switching overheads also do not arise when using spin locks in userspace. However, to avoid excessive spin delays, it is necessary to either react to the preemption of lock-holding tasks [19], or to prevent it entirely (with non-preemptive sections) [17, 18]. To enable the latter, LITMUS$^{\mathrm{RT}}$ has long supported low-overhead non-preemptive sections by means of a flag in memory between userspace and the kernel [3, 10], which inspired the use of the lock page in our protocols.

While a discussion of the relative merits of spin- and suspension-based synchronization is beyond the scope of this paper (*e.g.*, see [3, 10, 12, 17, 20]), it is interesting to note that, at a high level, the futex approach [16] can be understood as an attempt to realize (at least in the common case) one of the key advantages of spin-based protocols, namely the avoidance of mode-switches, while preserving the semantics and analytical properties of suspension-based protocols.

As already discussed in Sec. II-A, Franke *et al.* [16] were the first to present an implementation of futexes in the Linux

kernel. A detailed explanation of the futex API exposed by the Linux kernel, as well as how it can be used to build synchronization primitives, is provided by Drepper [15]. Interestingly, a conceptually similar, but less flexible implementation appeared previously in BeOS [26].

Züpke's work [29] on deterministic futexes targets the problem of providing a futex implementation in embedded separation kernels (*i.e.*, kernels that implement strict space and time partitioning). Züpke identifies problems with the applicability of the Linux approach, and proposes a method to implement futexes without the need for a kernel memory allocator. In subsequent work, Züpke *et al*. [30] proposed two implementations of futexes with IPCP semantics and evaluated their overheads. (Züpke *et al*. [30] also mention that such "fast" IPCP variants have been used in commercial RTOSs such as PikeOS [30] for a number of years.) Note that, in contrast to Züpke *et al*.'s work [30], our implementation realizes the *classic PCP* (*i.e.*, the OPCP), which minimizes priority inversion at runtime. Nonetheless, our implementation could be easily changed to implement IPCP semantics as well.

To the best of our knowledge, this is the first paper to present implementations and an evaluation of average-case-optimized versions of multiprocessor real-time locking protocols for partitioned schedulers.

VII. Conclusion

This paper demonstrates that the Linux kernel's futex approach [16] to constructing locking protocols—wherein expensive system calls are avoided when locks are uncontended, and which previously had been applied only to a reactive real-time locking protocol, namely the PIP—can be extended to anticipatory real-time locking protocols with stronger analytical properties such as the PCP, the MPCP, and the FMLP$^+$.

As expected, in an evaluation of our prototype implementations in LITMUS$^{\mathrm{RT}}$, we observed a substantial reduction in average- and worst-case overheads in the uncontended case (*e.g.*, the average cost of unlocking a semaphore under the PCP-DU-BOOL is 92% lower than in the baseline PCP implementation). However, we also observed increased worst-case overheads for some operations (*e.g.*, the maximum overhead under the PCP-DU-BOOL increased by 37%). While the latter is higher than we initially expected, it is no worse than Linux's existing PRIO_INHERIT futexes. Overall, due to the significant improvement in the uncontended case, futexes remain an attractive choice for workloads that can tolerate the increase in worst-case overheads. In particular, we believe that the improvements in common-case locking overheads are useful for the vast majority of predominantly soft real-time applications deployed on Linux and Linux-like platforms.

In future work, it would be interesting to investigate whether the increase in worst-case overheads can be avoided entirely with further optimizations. Further, it would be worthwhile to reassess the costs and benefits of futexes in the context of systems in which worst-case overheads are determined using static worst-case execution time analysis.

Acknowledgement. We thank Dr. Al Grant of ARM Ltd. for providing us with access to CoreSight tracing tools.

References

[1] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[2] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA'07*, 2007.

[3] B.B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[4] B.B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS'13*, 2013.

[5] B.B. Brandenburg. The FMLP$^+$: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *ECRTS'14*, 2014.

[6] B.B. Brandenburg and J.H. Anderson. Feather-trace: A light-weight event tracing toolkit. In *OSPERT'07*, 2007.

[7] B.B. Brandenburg and J.H. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *RTCSA'08*, 2008.

[8] B.B. Brandenburg and J.H. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS'10*, 2010.

[9] B.B. Brandenburg and J.H. Anderson. Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks. In *EMSOFT'11*, 2011.

[10] B.B. Brandenburg, J.M. Calandrino, A. Block, H. Leontyev, and J.H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *RTAS '08*, 2008.

[11] A. Burns and A.J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[12] A. Burns and A.J. Wellings. A schedulability compatible multiprocessor resource sharing protocol — MrsP. In *ECRTS'13*, 2013.

[13] J.M. Calandrino, H. Leontyev, A. Block, U.C. Devi, and J.H. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS'06*, 2006.

[14] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1), 1978.

[15] U. Drepper. Futexes are tricky. Available at http://www.akkadia.org/drepper/futex.pdf, 2005.

[16] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, 2002.

[17] P. Gai, M. di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In *RTAS'03*, 2003.

[18] P. Gai, G. Lipari, and M.D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01*. IEEE, 2001.

[19] L.I. Kontothanassis, R.W. Wisniewski, and M.L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, 1997.

[20] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS'09*, 2009.

[21] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[22] R. Müller, D. Danner, W. Schröder-Preikschat, and D. Lohmann. MULTI SLOTH: An efficient multi-core RTOS using hardware-based scheduling. In *ECRTS'14*, 2014.

[23] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS'11*, 2011.

[24] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS'90*, 1990.

[25] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. 1991.

[26] B. Schillings. Be engineering insights: Benaphores. *Be Newsletters*, 1(26), 1996. Archived copy available at http://www.haiku-os.org/legacy-docs/benewsletter/Issue1-26.html.

[27] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), 1990.

[28] T.S. Taft and R.A. Duff. *Ada 95 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652: 1995 (E)*, volume 8652. Springer, 1997.

[29] A. Züpke. Deterministic fast user space synchronization. In *OSPERT'13*, 2013.

[30] A. Züpke, M. Bommert, and R. Kaiser. Fast user space priority switching. In *OSPERT'14*, 2014.