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

Roy Spliet
MPI-SWS
rspliet@mpi-sws.org

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

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

Sven Dziadek
TU Dresden
dziadek@gmail.com

***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 locks under the Priority Inheritance Protocol (PIP) do not incur kernel-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 in these protocols and present the design and evaluation of their implementation under LITMUS$^{\mathrm{RT}}$, a real-time extension of the Linux kernel. Our evaluation shows that the average-case locking overhead is improved by up to 87% for some scenarios, but it does come at a price in the worst case overhead.**

## I. Introduction

Suspension-based real-time locking protocols, such as the Priority Inheritance Protocol (PIP), are used in real-time systems to ensure mutually exclusive access to shared resources while preventing unbounded priority inversions, where a higher priority task that should be scheduled can be delayed by lower priority tasks for potentially unbounded durations. Such blocking, termed *priority-inversion blocking* (henceforth pi-blocking), increases worst-case response times of tasks. Various protocols have been proposed which provide different bounds on worst-case blocking time. The choice of real-time locking protocol is thus critical in the design of real-time systems as it affects their schedulability.

Practical systems, in addition to accounting for pi-blocking, must also carefully account for system overheads associated with the 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. Ideally, one would choose an optimal locking protocol, such as the Priority Ceiling Protocol (PCP), provided that an efficient implementation is available.

The majority of prior work dealing with the efficient implementation of real-time locking protocols has focused on worst-case overheads. However, there are numerous workloads that can benefit from faster average-case overheads, *where locks are typically uncontended*. A prime example is that of soft real-time workloads such as video playback where locks may be acquired by threads of different priorities many thousands of times per second [7]. Reducing average-case locking overhead in such applications allows for supporting much higher throughput (*e.g.* higher frame-rate) and reducing the number of deadline misses. Another advantage is in mixed-criticality systems where optimizing for the average-case can help avoid deadline misses in low-criticality tasks. While these examples motivate the need for low average-case overheads, system schedulability is ultimately based on the *worst-case overheads*. Thus, average-case overhead reductions must not significantly increase worst-case overheads.

An effective approach for achieving these goals is through the support for the *fast userspace mutexes* (futexes) [11] in the Linux kernel, a mechanism that supports efficient lock implementations with low average-case overheads. By exporting lock-state information to userspace, expensive system calls can be avoided when a lock is uncontended. Linux's PIP implementation uses futexes to reduce the average-case locking overhead.

While prior work [1] has successfully applied the futex approach to the PIP, the approach does not easily generalize to more sophisticated protocols. This is problematic as the PIP is limited in its theoretical properties: it has non-optimal bounds on pi-blocking and is susceptible to deadlocks unlike the PCP. Additionally protocols such as FIFO Multiprocessor Locking Protocol (FMLP$^{+}$) are asymptotically optimal in clustered Job-Level Fixed-Priority scheduling, a property not shared by the PIP. However, it is unclear whether the futex approach can be extended to improve the average-case performance of these protocols. In fact, even though the *Pthreads* package provides an implementation of the PCP (called PRIO_PROTECT) which uses the futex API, the PRIO_PROTECT implementation does not adhere to the futex philosophy and requires multiple system calls for uncontended lock operations.

The key challenge in generalizing the PIP futex implementation to more sophisticated protocols boils down to the following difference: The PIP can be considered a *reactive* locking protocol, in that it makes all its decisions in reaction to lock acquisition or lock release operations. In contrast, the PCP and multiprocessor locking protocols such as MPCP and FMLP$^{+}$ are what we term *anticipatory* locking protocols: they make use of additional information about the workload to anticipate problematic scenarios and prevent them. For example, in the PCP, each lock is associated with a priority ceiling that specifies the highest priority task that may acquire it. This can result in scenarios where a lock is uncontended but the semantics of the protocol forbid a task from acquiring it. Similarly, MPCP and FMLP$^+$ keep track of additional information to ensure bounded *remote blocking* via the technique of *priority boosting*, where jobs in critical sections are boosted in priority to ensure progress.

This motivates the key question of this paper: 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? In this paper, we show that this is indeed possible and we describe the key properties that help us achieve this for three protocols: the PCP, MPCP and FMLP$^+$. We implemented futex-based versions of these protocols in LITMUS$^{\text{RT}}$ called the PCP-DU, MPCP-DU and FMLP$^+$-DU. Here, "DU" stands for *deferred update*, as our implementations implement futexes by deferring the update of the state of a lock until the next time the scheduler is called.

## A. Contributions

In this paper, we make the following contributions:

- We identify key properties of the PCP, MPCP and FMLP$^+$ (Secs. IV and V) that allow us to extend the futex approach to anticipatory real-time locking protocols.
- We describe futex-based implementations of these protocols (Secs. IV and V) in LITMUS$^{\text{RT}}$: the PCP-DU, MPCP-DU and FMLP$^+$-DU respectively.
- We evaluated these protocols (Sec. VI) and observe upto 93% reduction in overheads in certain uncontended scenarios, and only at most 22% increase in contended overheads.

We begin by providing the necessary background information before presenting the design, implementation details and evaluation of the improved implementations.

## II. Background and definitions

We consider real-time scheduling under a system comprising a set of $n$ tasks $\{T_1, \ldots, T_n\}$ running on a set of $m$ processors $\{P_1, \ldots, P_m\}$, where each task $T_i$ *releases* a series of jobs, where the $j^{th}$ job is denoted $J_{i,j}$. We assume the *sporadic task model* where each task is defined by the *worst-case execution time* (WCET) $e_i$, a minimum inter-arrival time or period $p_i$ and a relative deadline $d_i$. There are a set of shared resources $\in R = \{r_1, \ldots, r_n\}$ in the system, each associated with a lock.

We assume Partitioned Fixed-Priority (P-FP) scheduling [9], where the set of tasks are partitioned among the $m$ 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* ($T_1$ having the highest priority and $T_n$ having the lowest priority). Tasks are scheduled based on their *effective priority*, which may exceed the base priority under certain resource sharing policies. For instance, the PIP may temporarily raise the effective priority of a task.

It should be noted that although we use P-FP scheduling throughout this paper, the techniques presented are not limited to P-FP and generalize to other scheduling algorithms.

## A. Futexes

Futexes, short for *fast userspace mutexes*, are a mechanism found in the Linux kernel to support the implementation of suspension-based locks in userspace [11]. 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. The integer represents the number of tasks contending for this lock: 0 means free, 1 means acquired, and any larger value indicates there are jobs blocking on this mutex. Lock and unlock operations are implemented using atomic fetch-and-increment/fetch-and-decrement respectively, where the old value indicates whether the lock operation succeeded or some further action is required. Further action involves an expensive system call to the kernel in order to manipulate the wait-queue.

## B. Locking in LITMUS$^{\text{RT}}$

LITMUS$^{\text{RT}}$ is a real-time extension of the Linux kernel [8], providing a testbed for novel scheduling and locking algorithms. It supports the sporadic task model and modular scheduler plugins. Besides several multiprocessor scheduling policies, LITMUS$^{\text{RT}}$ also implements a variety of locking protocols, including the PCP, MPCP, FMLP [6] and the FMLP$^+$. These implementations have been designed primarily with low worst-case locking overheads in mind.

Real-time tasks use the mutex-based locking API of LITMUS$^{\text{RT}}$ to ensure mutually exclusive access to shared resources. Tasks use special system calls provided by LITMUS$^{\text{RT}}$ to create locks associated with a particular protocol, and to lock and unlock them. These routines track the state of a lock and take care of task blocking when the protocol forbids lock acquisition for a particular task. We implemented our protocols in LITMUS$^{\text{RT}}$ to benefit from the generic scheduling framework and extensible API it offers.

## C. Real-Time Locking Protocols

The simplest suspension-based real-time locking protocol is the Priority Inheritance Protocol (PIP) proposed by Sha *et al.* [14]. 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 unbounded priority inversions. PRIO_INHERIT is the Linux implementation of the PIP, which uses the futex API and consequently 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, which occurs when a *chain* of blocked jobs forms where each job blocks on a resource currently held by the next job in the (ascending priority-ordered) chain. The chain is resolved in reverse priority order, so that the highest priority task in this chain is blocked for the duration of all critical sections in this chain. Second, inconsistently ordered nested locks are susceptible to deadlocks under the PIP. These problems discourage using PIP from an analytical point of view.

### D. Priority Ceiling Protocol (PCP)

The *Priority Ceiling Protocol* [14] addresses the shortcomings of the PIP. It is optimal on uniprocessors and jobs experience pi-blocking of at most one outermost critical section. Furthermore, through precautionary blocking of jobs in anticipation of cyclic dependencies, deadlocks are avoided entirely.

In the PCP, every resource $r_i$ is assigned a *priority ceiling*, $ceil(r_i)$, which is the highest priority of all tasks that may lock that resource. During run-time, the scheduler keeps track of a *system ceiling* (denoted $sysceil(t)$), which is the currently held lock with the highest priority ceiling. A job is only allowed to acquire a lock if *(i)* either its priority exceeds the priority of the system ceiling, or *(ii)* if the system ceiling is currently owned by the requesting job. If none of these conditions are satisfied, 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 to the highest priority waiter, who then becomes the new owner of the system ceiling.

LITMUS$^{\text{RT}}$ implements the *Classic* or *Original* PCP (OPCP), where 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 the lower priority job is raised only to that of the highest priority job currently blocked by it.

The *pthreads* implementation of the PCP on the other hand (also known as PRIO_PROTECT) is an implementation of the *Immediate Priority Ceiling Protocol* (IPCP), which is subtly different from the OPCP. The difference to the OPCP is that when a job acquires a lock, its priority is immediately raised to the priority ceiling of that lock using priority inheritance. While this provides less accurate control over the effective priority, it has the advantage of reducing the number of preemptions in the system and thus the runtime overhead.

Although PRIO_PROTECT is implemented in Linux using the futex API, it does not enjoy the benefits of the futex approach. Anticipatory properties are not captured in the futex API, which for the PCP implies a system call is still required to raise or lower the jobs' priority, even in the uncontended case. PRIO_PROTECT thus does not enjoy the benefits of the futex approach for uncontended lock operations.

### 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 is blocked by other jobs on different processors. Unfortunately the progress guarantees of the PIP, which is also a key component of the PCP, breaks across partition boundaries and does not ensure the progress of resource-holding jobs.

Rajkumar *et al.* proposed the *Multiprocessor Priority Ceiling Protocol* (MPCP) [13], the first shared-memory multiprocessor real-time locking protocol, which solves the problems associated with parallelism through *priority boosting*. Priority boosting is a technique where the priority of jobs in critical sections is temporarily raised to a level higher than any used base priority. These jobs are thus guaranteed to progress as they cannot be interrupted by newly released jobs (which have not entered a critical section yet), but boosted jobs can still be preempted by other jobs in the boosted state. Under the MPCP, for every partition in the system, each lock is assigned a priority ceiling, which is the highest priority of all tasks on every other partition accessing this lock. 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.

Deadlocks are avoided in the MPCP by prohibiting the nesting of locks and waiting jobs are ordered in order of their base priority; when a critical section completes, jobs blocked on that lock are resumed in order of priority.

LITMUS$^{\text{RT}}$ implements the MPCP using the same APIs as the PCP. Upon opening a lock, its priority ceilings for all other partitions in the system are set up. When the job tries to acquire the lock, its priority is boosted and an attempt is made to obtain the lock. On failure, the job is added to the priority wait-queue associated with the lock and is suspended. When the lock is unlocked, the job is restored to its base priority and the next job on the wait-queue is resumed.

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

The *FIFO Multiprocessor Locking Protocol* (FMLP$^+$) [5] is a suspension-based 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 crucial difference is that it uses FIFO ordering for jobs in critical sections as well as in the wait-queue. Similar to the MPCP, nesting of critical sections is forbidden, thus avoiding deadlocks.

The FMLP$^+$ implementation in LITMUS$^{\text{RT}}$ differs slightly from the MPCP: 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 instead of the ceiling of the locks.

Having described the necessary background, we now present the key insights of three protocols, namely the PCP, MPCP, and FMLP$^+$, that allow us to implement futex-based versions, along with their implementation under LITMUS$^{\text{RT}}$.

## III. Design and Implementation

In the following two sections, we identify the challenges in designing futex-based versions of three real-time locking protocols: the PCP, MPCP, and FMLP$^+$. We chose these protocols because these are the base implementations in LITMUS$^{\text{RT}}$, and are known to be state of the art for uniprocessors and multiprocessors respectively.

In the following two sections, for each of these three protocols, we identify scenarios that allow for optimization of the uncontended case. At a high level, we point out scenarios that allow us to safely defer the updating of the global lock and scheduler state until later (specifically, the next context switch), without violating the semantics of the protocol. We then present the design and implementation of our futex-based versions (the PCP-DU, MPCP-DU, and FMLP$^+$-DU respectively), with the goal of reducing the overhead of uncontended lock and unlock operations, while minimizing the impact on the worst-case overhead.

We begin with the PCP and then apply what we learn there to the MPCP and the FMLP$^+$.

## IV. Uniprocessor Protocols - PCP

As explained in Section II-D, the PCP is an anticipatory uniprocessor suspension-based locking protocol where each lock has a priority ceiling, defined as the highest priority of all tasks using that lock. Acquiring locks potentially raises the system ceiling, preventing other tasks with a priority below the system ceiling from acquiring any locks. Similarly, releasing locks may potentially lower the system ceiling resulting in blocked tasks waking up. Correctly performing these operations requires an accurate method of keeping track of the system ceiling at any given time which requires a global view of the state of all the locks in the system.

The current implementation of the PCP in Linux (PRIO_PROTECT) maintains the status of all locks inside the kernel and update this on every lock and unlock operation. This works as each of these operations are performed through a system call, allowing the kernel to track the system ceiling. However, this is at odds with the goal of allowing userspace processes to modify lock state without any kernel intervention, as is the case with futexes. In order to support this, processes must be able to communicate this state to the kernel. Further, this state should only be communicated to the kernel if and when strictly needed in order to avoid unnecessary overhead.

We first identify two key properties of the PCP that allows us to build such an "asynchronous" locking system: *(i)* the success or failure of lock acquisition by a particular job can be determined prior to it being scheduled, and *(ii)* since the PCP is a uniprocessor locking protocol, the state of a lock (*i.e.* whether it is locked by someone or not) needs to be known only when a context switch occurs. These two properties together allow us to implement futexes for the PCP.

These properties are derived from the observation that in the PCP, the outcome of a lock operation depends only upon the effective priority of the job and the current system ceiling, both of which are known by the scheduler when a job is about to be scheduled. Further, since the PCP is a uniprocessor protocol, neither of these two factors can change without the scheduler being invoked as the only events that trigger changes in either of these are always preceded by a context switch. For example, a higher priority task may be released which acquires a lock, but not before preempting the old task, thus invoking the scheduler. The same rationale is true for unlocking: whether

```
struct lock_page {
    bitmap_t locked;
    bool can_lock;
    int unlock_syscall;
};
```

Listing 1: Structure of the lock page

or not jobs are blocking on a system ceiling will not change within a scheduling interval of the system ceiling owner.

This immediately tells us that the latest time until which we can defer communication of the state of locks back to the kernel is a context switch. More importantly, it is possible to predict the outcome of the lock and unlock operations during an interval from when a job is scheduled to the time it is preempted and the scheduler invoked.

### A. Implementation of PCP-DU

To implement futexes, all that is needed is a bidirectional communication channel between a userspace process and the kernel to exchange lock state information. Userspace processes need to be able to convey the state of locks to the kernel on being preempted. This can be done by a single bit for each lock in the system, which is set by the userspace process and can be later read out by the kernel when needed (*i.e.* when a context switch occurs).

In addition, the kernel needs to be able to communicate two pieces of information. First, whether the userspace process can successfully acquire locks and second, whether an unlock operation requires scheduler invocation in order to wake up blocked higher-priority tasks. If the kernel is able to communicate this information to every task, system calls can be avoided during lock and unlock operations unless strictly needed.

In our implementation, called PCP-DU standing for PCP with deferred updates, each task shares a memory page with the kernel with the structure shown in Figure 1. This shared page acts as the bidirectional communication channel between the task and the kernel. Here, the *locked* bitmap stores the state of each lock the task may hold, the *can_lock* bit is set by the kernel to communicate to a task whether it may acquire locks without kernel intervention, and the *unlock_syscall* field is a reference counter specifying whether there are higher-priority tasks blocked on a lock or not. In case of a non-zero value for *unlock_syscall*, the task should invoke the scheduler upon releasing a lock.

PCP-DU requires two additions to the existing scheduler: updating the global state of the locks touched by the current job and determining whether the next job can successfully acquire locks without kernel preemption or not. For updating the global state of the locks, the scheduler compares the `locked` bitmap from the current job's lock page with a previously cached copy to determine relevant events in the past interval. It then updates the global lock state accordingly. The kernel updates this cached copy of the *locked* bitmap on every context switch. After updating the lock states, the scheduler determines whether the next job is allowed to lock by comparing its effective priority with the priority of the system ceiling. As mentioned earlier, this can be communicated to the process using the the *can_lock* bit in the lock page.

*a) **Lock acquisition**:* Lock acquisition consists of first checking the *can_lock* boolean. If it is unset, then the scheduler is invoked through a system call which blocks the process. If the *can_lock* boolean is set, the task need simply write to the bit in the *locked* bitmap corresponding to the lock it wishes to acquire. It should be noted at this point that checking the *can_lock* boolean and locking a bit in the bitmap must be done atomically. Otherwise, a preemption occuring right after the *can_lock* boolean is checked can result in an invalid lock operation. This atomicity can be achieved by dedicating one bit in the bitmap to the *can_lock* field.

In addition to the approach described above (called PCP-DU-BOOL), we also propose a novel approach using the exception-handling mechanism available in the memory management unit (MMU) of modern systems to communicate the *can_lock* bit. In this approach, the kernel sets the permission of the lock-page to read-only if the process is not allowed to lock. Lock acquisition thus consists of attempting to write a bit to the bitmap. The lock is acquired when this write operation succeeds, otherwise an access violation fault is triggered which automatically invokes the kernel, allowing it to block the requesting process. This implementation, called PCP-DU-PF, has the advantage that locking consists of writing only a single bit in memory. There is no for atomic operations and the code can be made sequential, thus avoiding branch mispredictions.

*b) **Lock release**:* For unlocking in userspace, the kernel needs to efficiently determine whether a system call is required. We propose an implementation where atomicity is not required between unlocking (unsetting the desired bit in the *locked* bitmap) and checking whether the kernel should be invoked for unlocking jobs. As an added benefit, we trivially update the decision every time a job blocks, thus simplifying the logic needed at every context switch.

The basis for this mechanism is the *unlock_syscall* field, a reference counter that indicates whether the job should invoke the scheduler during an unlock operation. When a job blocks on the system ceiling, its owner's *unlock_syscall* field is incremented. On unlocking, this integer is read out. If zero, no jobs are blocked on one of the locks of the current job, and thus no action is required. However, a non-zero value indicates blocked higher-priority tasks and the task invokes the kernel. At this point, the kernel wakes up every task now exceeding the system ceiling, and the number of woken up jobs is subtracted from the integer.

Atomicity between unlocking and checking the *unlock_syscall* integer is not necessary, because in the rare case of preemption the kernel will update the global lock state based on the lock page and wake up any jobs required, updating the unlock_syscall integer. In the absolute worst case, the kernel will be invoked while the work was already done. This causes some overhead, but not incorrect behaviour.

We acknowledge that this mechanism does not keep track of which acquired lock(s) define the system ceiling. In the uncommon case of contended nested locks, it could happen that an unlock does not lead to a lower system ceiling. Implementation of a system that accounts for this corner case defeats the goals for this project; the more complex logic required would increase both the average- and worst-case lock time in return for efficiency of an uncommon case.

```
struct lock_page {
        bitmap boost,
        bool unlock_syscall
};
```

Listing 2: Structure of the lock page

## V. Multiprocessor Protocols - MPCP and FMLP$^+$

While with the PCP we were able to defer the update of the global system state until the next context-switch, this is not entirely possible with the MPCP and FMLP$^+$. Multiprocessor locking protocols suffer from *remote blocking*, where a local job may be blocked by jobs on a different processor. While in the PCP, a switch to another job was always preceded by a context switch, this is no longer the case since a lock may be acquired by a job running in parallel on a remote processor.

Both the MPCP and FMLP$^+$ use a technique termed *priority boosting* to ensure progress of critical sections. Priority boosting works by increasing the priority of jobs in critical sections beyond that of any local job, thus ensuring that they can never be preempted. Recall that in partitioned scheduling, as long as the priority is boosted past that of the highest-priority local job not within a critical section, we can guarantee that the boosted job will not be preempted by them. However, since boosted jobs are free to preempt other boosted jobs, each lock is assigned a priority ceiling which is the highest priority of all tasks on every other partition accessing this lock.

The key point is that priorities are only relevant within the same partition [5]; the difference in priorities of two jobs running on different partitions is meaningless. As the boosted priority also has no effect on the order a lock is granted under the MPCP (which uses the base priorities of the jobs) or the FMLP$^+$ (which uses the time of the request), any absolute or relative variation in boost priorities across processor boundaries has no effect on scheduling.

Thus, we observe that the boosting of priority, similar to priority inheritance, can be deferred until a context-switch occurs. This is because the preemption of a job is always preceded by a context-switch, at which point the kernel can observe that the job "boosted itself" and perform the actual boosting of priority. That is, it is safe to defer the actual boosting of a job's priority until a context-switch occurs.

### A. Implementation

For MPCP-DU and FMLP$^+$-DU, we use the same lock-page communication channel as for PCP-DU. The structure of the lock page differs slightly, as shown in Listing 2.

Recall that in order to implement deferred priority boosting, the job needs to be able to communicate to the kernel as to whether it boosted itself or not. This is implemented in MPCP-DU and FMLP$^+$-DU through the *boosted* bitmap, which represents the locks that should be considered for boosting (*i.e.* the job plans to acquire them). The boosted priority is the highest priority ceiling of all locks a job has set to the boosted state.

It should be noted that since a lock may get acquired on a remote processor without a local context-switch preceding it, we cannot store the state of the lock in the per-process structure anymore. Instead this state is shared among all processes in the form of a single shared page of memory comprising a reference counter. Remote blockers make their presence known by attempting an atomic increment of the lock. If it is currently acquired, they return non-zero and block. When the lock is released by the owner, they will notice that the reference count is non-zero and invoke the scheduler (which proceeds to wake up the remote blocked job).

The *unlock_syscall* is again used by the kernel to communicate to a task when an unlock operation, which results in the priority being lowered again, should call the kernel. This may occur if there are multiple boosted jobs and the highest priority one unlocks. The resulting drop in priority of that job should result in the next boosted task being scheduled.

*c) **Kernel operations during preemption**:* On every scheduling operation, the kernel will perform two additional actions atop the regular scheduling: *(i)* determine the priority of the current task, and *(ii)* determine whether the next task should invoke the scheduler upon unlocking.

For determining the priority of the *current* job, the kernel compares its `boost` variable with a cached copy obtained during the previous context switch. When there is a difference, the priority is updated accordingly. For FMLP$^+$-DU, the most significant bit in the `boost` bitmap will result in a new boost priority being set.

To determine whether the next task should invoke the scheduler on an unlock, the scheduler compares the *base* priority of the next scheduled job with any other *effective* priority on the local ready-queue. If there exists a job with a higher effective priority, the `unlock_syscall` flag on the job's lock page is set.

*d) **Lock acquisition**:* For lock acquisition in MPCP-DU and FMLP$^+$-DU, as mentioned before, we have to extend futex with a mechanism that supports deferred priority boosting. For this the `boost` bitmap is added to the shared lock page. Like the `locked` bitmap in PCP-DU, each bit corresponds with a single lock. When a job intends to acquire a lock, it first sets the boost bit. On a context switch these bits are interpreted and the boost priority is set accordingly.

For FMLP$^+$-DU the priority boosting mechanism must be able to distinguish between a critical section that lasted during the entire interval a job was scheduled, and the case in which the job releases and re-acquires the lock within this interval. In the latter case a new boost priority must be determined. For this purpose the most significant bit of the `boost` bitmap is set by the FMLP$^+$-DU implementation on every lock operation. The scheduler clears this bit before scheduling a job, helping it detect whether any resource was locked during the scheduled interval under the assumption locks are not nested as defined in the FMLP$^+$ specification.

Interrupts between setting the boost-bit and obtaining the lock can not result in incorrect behaviour. Preemption cannot occur on this interrupt, because the priority of the job already exceeds all local base priorities. It can thus continue the locking operation unless a job with a higher boost priority is unblocked on a semaphore. In this latter case the job is correctly preempted.

*e) **Lock release**:* For releasing a lock under the MPCP and FMLP$^+$, the boosted priority is dropped. In some cases this should lead to preemption even if the mutex being unlocked is not contended. We thus need to extend the futex mechanism with a check whether the scheduler should be invoked on an uncontended unlock.

The `unlock_syscall` field in the lock page is a boolean that conveys this information. When a futex unlock is uncontended, this bit is tested and when set the `boost` bit is unset after which kernel scheduler is called.

For lock operations that follow the slow futex path, unboosting is handled immediately by the kernel prior to possible preemption to guarantee a correct scheduler decision. In this case userspace does not have to take any further action to unboost the priority.

### B. Further Optimizations

In addition to the presented techniques, we have identified two performance optimizations for the implementation of the MPCP and the FMLP$^+$ in LITMUS$^{RT}$ that improve performance in both the average- and worst-case.

For both these protocols, a spinlock is used inside the kernel to protect the state of the per-mutex wait-queue. During lock acquisition, the critical section consists of adding the current job to the wait-queue. Initially the unlock critical section consisted of taking the first element off the wait queue and waking it up using `wake_up_task()`. We have observed that `wake_up_task()` can take several thousand cycles due to raising an inter-processor scheduling interrupt. By moving this routine outside the critical section, any lock operation blocked on this semaphore no longer waits for this operation to complete. This significantly reduces the worst-case response time for contended lock acquisition.

Second, in the FMLP$^+$ implementation in LITMUS$^{RT}$ the boost priority is set using the system clock to ensure a fair timestamp. Recall that for FMLP$^+$ only the order of assignment within a partition is used for scheduling decisions. We can thus replace the system-clock based timestamp with a per-processor logical timestamp that is incremented on every read. This eliminates part of the overhead associated with every priority-boost operation.

## VI. Experiments and Results

In this section we present the results from our evaluation of the different implementations: PCP-DU-PF, PCP-DU-BOOL, MPCP-DU and FMLP$^+$-DU. We compare these with the LITMUS$^{RT}$ implementations of the original PCP, MPCP

| (cycles) | PRIO_INHERIT | PRIO_PROTECT | PCP | PCP-DU-PF | PCP-DU-BOOL | MPCP-ORIG | MPCP-NEW | MPCP-DU | FMLP$^+$-ORIG | FMLP$^+$-NEW | FMLP$^+$-DU |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Samples: 7.100.000 per protocol | | | | | **Uncontended** | | | Samples: 12.200.000 per protocol | | | |
| Lock (Avg.) | 263 | 1604 | 645 | 160 | 171 | 1075 | 1091 | 214 | 1363 | 1041 | 215 |
| Lock (99%) | 351 | 1847 | 742 | 231 | 239 | 1300 | 1329 | 309 | 1594 | 1271 | 313 |
| Lock (Max.) | 7707 | 33156 | 3850 | 2384 | 1974 | 5875 | 6506 | 2477 | 7491 | 7906 | 2625 |
| Unlock (Avg.) | 301 | 1221 | 1211 | 87 | 92 | 1216 | 1149 | 177 | 1181 | 1117 | 174 |
| Unlock (99%) | 369 | 1475 | 1355 | 133 | 140 | 1451 | 1368 | 228 | 1418 | 1325 | 224 |
| Unlock (Max.) | 16552 | 9027 | 6655 | 1770 | 1653 | 7330 | 6158 | 1920 | 6574 | 5292 | 1875 |
| Samples: 900.000 per protocol | | | | | **Contended** | | | Samples: 2.200.000 per protocol | | | |
| Lock (Avg.) | 4955 | - | 1967 | 4624 | 2673 | 1266 | 1216 | 1097 | 1440 | 1096 | 1059 |
| Lock (99%) | 5550 | - | 2132 | 5367 | 3103 | 1531 | 1494 | 1387 | 1715 | 1371 | 1307 |
| Lock (Max.) | 7707 | - | 7953 | 14820 | 9700 | 12242 | 6281 | 5392 | 13109 | 6026 | 5570 |
| Unlock (Avg.) | 3767 | - | 4109 | 5148 | 4868 | 4301 | 4280 | 2657 | 4203 | 4166 | 2335 |
| Unlock (99%) | 6023 | - | 4627 | 5909 | 5582 | 5791 | 5799 | 5009 | 5741 | 5662 | 2962 |
| Unlock (Max.) | 13172 | - | 14919 | 15985 | 15787 | 20482 | 21657 | 18258 | 21223 | 22451 | 17991 |

TABLE I
OBSERVED AVERAGE AND WORST-CASE RESPONSE TIME FOR UNCONTENDED LOCK/UNLOCK OPERATIONS

and FMLP$^+$ and implementations of the MPCP and FMLP$^+$ containing the proposed optimizations, along with the standard Linux implementations PRIO_INHERIT and PRIO_PROTECT.

All our experiments were conducted on the Boundary Devices Sabre Lite ARMv7 development board based on the FreeScale I.MX6Q SoC, an ARM Cortex-A9 quad-core system running at 1GHz. All experiments were run using LITMUS$^{RT}$ 2013.1 based on Linux 3.10.5. Several bugfixes and performance improvements for spinlocks on ARM were backported into the 2013.1 tree from LITMUS$^{RT}$ 2014.1 and Linux 3.13 respectively. The kernel was compiled to the Thumb-2 instruction set with all kernel debugging options disabled in order to maximize performance.

In this section, we seek to answer the following key questions regarding our improved implementations:

1) How does the uncontended overhead in our implementations compare with those of the original protocols?
2) What additional overheads are introduced by the futex-based approach and how significantly does it affect the worst-case overheads?
3) Is it worth avoiding atomic operations when implementing uniprocessor locking protocols?

To answer these questions, we implemented microbenchmarks that measures lock and unlock overhead using hardware cycle-counters. Next we explain the methodology used in this microbenchmark.

## A. Microbenchmarking Methodology

Our microbenchmark program spawns several threads, each locking and unlocking a resource every period. Threads were randomly assigned a critical section length $C(T_i)$ between $60 - 80\mu s$, an execution time between $C(T_i) + 0 - 20\mu s$ and a period between $600 - 800\mu s$. For the uniprocessor protocols 5 threads are spawned on one core, whereas for the multiprocessor protocols 6 threads are spawned across three cores. The fourth is reserved for trace and control tasks. A cache-polluting background workload was executed during and around the critical section to simulate a realistic workload. This program was ran for 5 seconds, repeated 300 times for each protocol with a different random parameter assignment for each run.

Each sample was measured using cycle counters which measured the time it takes from a lock or unlock request being made to when the operation completes (and the program continues). In the case of contended locks and preempted unlocks, we subtract the blocking time as this is a consequence of application parameters as well as other system overheads such as the time it takes to wake up a task on a remote processor, which we present separately.

To prevent any bias in the worst-case results, we normalized the size of the obtained data set across each class of protocols (uniprocessor and multiprocesssor) by randomly discarding samples from the data set. Final data set sizes are stated in Table I. For the scheduler overheads 16 million samples were obtained.

Separation of contended and uncontended lock samples is done at the first point of testing the lock status. For the locks proposed in this paper this is the atomic test in userspace. Race conditions that change the actual lock status from contended to uncontended beyond this point thus still result in a contended lock sample.

The results of our overhead analysis are shown in Table I. The table shows the overheads from the uncontended case in the top half of the table and the overheads from contended cases in the bottom half. For each case, the average, $99^{th}$ percentile, and maximum observed overheads are shown for both the lock and unlock operations. PRIO_INHERIT and PRIO_PROTECT is the Linux futex-based implementation of the PIP and the non-futex implementation of the PCP respectively. PCP refers to the LITMUS$^{RT}$ non-futex implementation while PCP-DU-BOOL and PCP-DU-PF refer to our two implementations that make use of atomic operations and the MMU exception handler respectively. MPCP-ORIG and

FMLP$^+$-ORIG refer to the default implementations of the MPCP and FMLP$^+$ found in LITMUS$^{\mathrm{RT}}$, while MPCP-NEW and FMLP$^+$-NEW refer to our optimized implementations. MPCP-DU and FMLP$^+$-DU refer to our futex-based implementations of the MPCP and FMLP$^+$.

### *B. Uncontended-Case Overhead Reduction*

The first question we wish to answer is to what extent the futex approach improves the average-case lock and unlock overheads. As can be seen in Table I, in the uncontended case, the overhead of the futex-based implementation with their vanilla counterparts shows overhead reductions of up to 13x (1200+ cycles for PCP unlock vs. only 90 cycles for PCP-DU-PF and PCP-DU-BOOL). This trend in uncontended overheads can be observed in both lock and unlock operations for every protocol when compared with their futex-based implementation (in both average and worst cases). This large reduction in average overhead is a direct consequence of the futex-based approach avoiding system calls when a lock is uncontended.

Note that the $99^{th}$ percentile value for all the uncontended results are very close in value to the observed average cycles. For example, the average locking time in PRIO_INHERIT is 263 cycles while the $99^{th}$ percentile is still only 351 cycles. This is in contrast to the observed maximum overhead of 7707 cycles. This shows that the futex-based approach actually performs really well on average, which is what we are concerned with in this paper.

### *C. Contended-Case Overhead Penalty*

Recall that in all three protocols, both lock and unlock code paths are modified to now do additional work. First, we added additional logic to the user-space lock routines, which for contended lock acquisition serves no purpose. Second, the scheduler has been extended to handle deferred update.

Thus, the next question we naturally ask, which ties back to our original goals, is whether there is a significant increase in the maximum observed overheads as a result of the additional logic we add to implement futexes for each of these protocols.

Table I provides some data that helps to answer this question. Note that we do not have contended samples for PRIO_PROTECT as it is an implementation of the IPCP (see Sec. II), where the priority of the requesting task is immediately raised to the priority ceiling of the resource. As a consequence, there can be no contention unless there is nesting, a scenario we do not evaluate in our microbenchmark.

Ignoring the average and $99^{th}$ percentile for now, let us consider the maximum observed overhead for lock and unlock operations under PCP. In this case, we see the maximum for the lock operation is 7953 cycles while the unlock is around 14919 cycles. If we compare this with our futex-based implementations (PCP-DU-PF and PCP-DU-BOOL), we can deduce two interesting insights. In the case of PCP-DU-BOOL, we see an increase in locking overhead of 1750 cycles (9700 cycles maximum in PCP-DU-BOOL vs. 7953 cycles in vanilla PCP). The unlock operation incurs an additional 900 cycles of overhead (increases from 14900 cycles to 15800 cycles). This is an additional overhead of 22% in the case of the lock operation while it amounts to around 6% for the unlock operation. This overhead is higher than we expected and we attribute the increase to the cumulative effect of atomic operation failures as well as cache pollution during job suspension. Nevertheless, since the reduction in the average-case overheads is significant, it is still beneficial to workloads that can tolerate the increase in worst-case overheads.

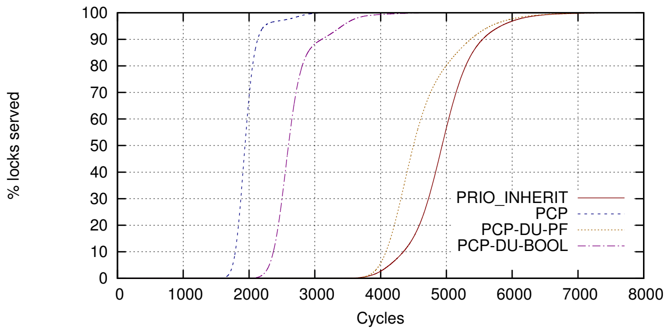

Fig. 1. Cumulative contended lock time (uniprocessor)

An interesting point to note regarding PCP-DU-PF is that the maximum observed overhead in the case of lock almost doubles (increases to 14,820 cycles from 7,953 cycles). We attribute this large increase in overhead to the complex code path of Linux's page-fault handling mechanism which is not optimized for our futex implementation. A more lightweight kernel might however find PCP-DU-PF more suitable given the large overhead reduction in the uncontended case. In any other case, our results suggest that it is not worth avoiding atomic operations, in particular on uniprocessors where there is no overhead from cache-coherency protocols.

Note that in the case of MPCP-DU and FMLP$^+$-DU, the maximum observed overhead actually *decreases* in comparison with the non-futex implementation in LITMUS$^{\mathrm{RT}}$ as the optimizations we describe in Sec. V push some of this overhead into the scheduler. As we show later in Sec. VI-E, this overhead is low and does not invalidate our results.

Importantly, the data in Table I shows an incomplete view of where the overheads come from. Figs. 1 and 2 help better understand which code paths increase the overhead and what percentage of samples they really affect.

### *D. Overhead Distributions*

Figs. 1–4 visualize the cumulative distribution of lock and unlock overheads, normalized to percentage. The X-axis shows the number of cycles while the Y-axis shows the percentage of locks served in less than the corresponding X-axis value.

From these graphs we attempt to recognize the different code paths in our implementations through the presence of horizontal shifts (which means, jobs experienced this code path during their execution after a certain percentage of operations).

For all the uniprocessor protocols, the contended locking code path is simpler, resulting in smooth curves as can be seen in Fig. 1. For the PCP and the PCP-DU, the slight anomaly

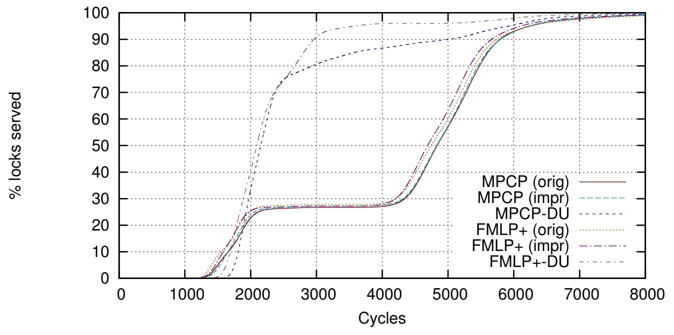

Fig. 2. Cumulative contended unlock time (multiprocessor)

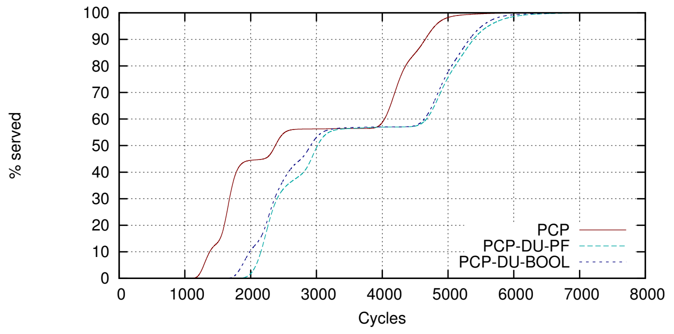

Fig. 3. Cumulative contended scheduling time (uniprocessor)

in the upper 10% of the slowest lock acquisitions corresponds with the case that an active old priority inheritance must be cleared prior to setting a new one.

From Fig. 1, we can derive that the PCP-DU-BOOL requires approximately 800 extra cycles for a contended lock acquisition than the PCP, evident from the right-shift distance between the curves. This overhead can be attributed to both the userspace code and the extra work involved in setting permission on the lock page. PCP-DU-PF has a significantly higher overhead in this case, spending more than twice the amount of cycles in comparison to the PCP. A lot of this overhead is due to the page-fault handling code path in Linux of which only a fraction is for PCP-DU-PF. Nonetheless, all implementations are more efficient than the PRIO_INHERIT code in *pthreads*.

Another example we show is the unlock operation overhead in the multiprocessor protocols. Fig. 2 clearly distinguishes two code paths all non-futex based protocols. In the implementation, the lower 25% of overheads correspond to the cheap code path where a job is woken up locally, while a more expensive code path can be seen in the non-futex protocols corresponding to when a job needs to be woken up remotely through an inter-processor interrupt. Due to changes in the distribution and adding a third code path to the unlock logic for MPCP-DU and FMLP$^+$-DU, the expensive code path hardly ever occurs.

The complete set of cumulative overhead results can be found in Appendix C.

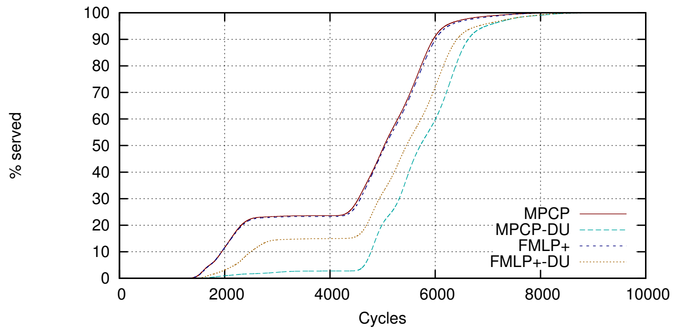

Fig. 4. Cumulative contended scheduling time (multiprocessor)

## E. Scheduler Overheads

Figs. 3 and 4 show the cumulative distribution of scheduler overheads similar to the previous graphs of lock and unlock overheads. In Fig. 3, we first observe that the scheduler overhead for PCP-DU-BOOL and PCP-DU-PF are mostly equal, despite the latter requiring a TLB flush when the locking permission changes. Significantly, as can be seen by the same shape of all the curves, both PCP-DU-BOOL and PCP-DU-PF only incur a fixed additional overhead in comparison to the original PCP of 550 cycles.

For multiprocessor protocols, we observe a different trend as can be seen in Fig. 4. We observe that for the majority of the cases, the MPCP-DU is more expensive in the scheduler than the FMLP$^+$-DU. This is expected, because unlike for the FMLP$^+$, the boost priority is not dependent on the specific lock, thus deciding whether the priority must be boosted or not requires less work.

Compared to the baseline implementations, overall the FMLP$^+$-DU is approximately 450 cycles more expensive on scheduling, and an additional 200 cycles is incurred by the MPCP-DU. We consider these good results, as this is considerably less than the number of cycles saved on a single uncontended lock and unlock operation.

## VII. Related work

Sha *et al*. identified the priority inversion problem and proposed the Priority Inheritance Protocol (PIP) to bound pi-blocking, and the optimal Priority Ceiling Protocol (PCP) to prevent deadlocks and limit pi-blocking to the length of at most one outermost critical section [14]. Evolving on the PCP, Baker *et al*. proposed the Stack Resource Policy (SRP) [2], achieving the same properties as the PCP by delaying job release to avoid blocking during execution.

Rajkumar *et al*. proposed the first suspension-based locking protocols for multiprocesor systems [13]; the Distributed Priority Ceiling Protocol (DPCP) and the Multiprocessor Priority Ceiling Protocol (MPCP) extended the PCP for distributed and shared-memory systems respectively under partitioned fixed-priority scheduling.

Global FMLP, proposed by Block *et al*. [3], was the first suspension-based locking protocols for global earliest-deadline first scheduling. Although we focus on partitioned algorithms in this paper, the work on FMLP$^+$ was a refinement of the FMLP. The *FIFO Multiprocessor Locking Protocol* (FMLP$^+$) [5] is a suspension-based locking protocol for partitioned scheduling with an asymptotically optimal bound on maximum suspension-aware pi-blocking ($O(n)$).

The observation that kernel calls are expensive is not new. Liedke *et al*. [12] establish that the kernel overhead is prohibitive for efficient IPC, and propose a solution in userspace that reduces IPC overhead while guaranteeing atomicity.

Franke *et al*. [11] observe the same kernel cost for synchronization primitives, and proposed *fast userspace mutexes* (futexes) as a general mechanism for implementing lightweight synchronization which issues system calls only when locks are contended. Futexes are part of the Linux kernel userspace API and Drepper [10] provides a detailed explanation of the futex API exposed by the Linux kernel and how it can be used to build synchronization primitives. The futex API has been used to implement lightweight PIP futexes (PRIO_INHERIT in the *pthreads* package) [1]. Our work is motivated by the elegance of this approach and we extended it to support anticipatory protocols with more attractive analytical properties (in particular the PCP, MPCP, and FMLP$^+$). Züpke's work [15] on deterministic futexes targets the problem of futex implementation from the perspective of small embedded real-time kernels, identifies problems with the applicability of the Linux approach, and proposes a method to implement futexes without the need for a fine-grained kernel memory allocator.

LITMUS$^{\mathrm{RT}}$, the real-time extension of the Linux kernel developed at UNC Chapel Hill provides implementations for many state-of-the-art locking protocols[6], [8] and while there has been prior work on analysing the overheads of locking protocols in LITMUS$^{\mathrm{RT}}$ [4], the focus has been on improving the worst-case response time. To the best of our knowledge, this is the first paper to implement and evaluate average-case-optimized versions of the PCP, the MPCP and the FMLP$^+$.

## VIII. Conclusion

Motivated by the benefits of improving the average-case overheads of locking protocols, this paper explored whether it is possible to extend the futex approach of the Linux kernel, where expensive system calls are avoided when locks are uncontended, to anticipatory locking protocols which provide stronger analytical properties.

We identified key properties of three protocols (the PCP, MPCP, and FMLP$^+$) that leave room for optimization of the overhead in the uncontended case, and successfuly implemented and evaluated futex-based version of these protocols under LITMUS$^{\mathrm{RT}}$.

Our cycle-accurate evaluation of these protocols shows up to 87% reduction in the uncontended case for certain scenarios while incurring no more than 22% additional overhead in the worst-case. While this is higher than expected, the significant reduction in uncontended overheads still make this attractive for workloads that can afford the increase worst-case overhead.

Overall, we have shown that it is possible to implement state-of-the-art real-time locking protocols in the spirit of the futex approach. We have observed compelling improvements for the average-case uncontended lock operations. However, we have also observed that the worst-case overheads must not be neglected. In fact, due to the moderate increase in maximum overheads, our futex approach may not be suitable for workloads that exhibit high contention or are especially sensitive to worst-case lock overheads. However, we believe that the improvements in average-case locking overheads are useful for the vast majority of predominantly soft real-time applications deployed on Linux and Linux-like platforms.

## References

[1] Lightweight priority inheritance futexes. http://lxr.linux.no/linux/Documentation/pi-futex.txt.

[2] T.P. Baker. A stack-based resource allocation policy for realtime processes. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 191–200, 1990.

[3] Aaron Block, Hennadiy Leontyev, Bjorn B. Brandenburg, and James H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 47–56, Washington, DC, USA, 2007. IEEE Computer Society.

[4] B.B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for p-fp scheduling (extended version). In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 141–152, April 2013.

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

[6] Björn B. Brandenburg and James H. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '08, pages 185–194, Washington, DC, USA, 2008. IEEE Computer Society.

[7] Björn B. Brandenburg and James H. Feather-trace: A light-weight event tracing toolkit. In *In Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'07*, pages 61–70, 2007.

[8] 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 *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 111–126, Dec 2006.

[9] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[10] Ulrich Drepper. Futexes are tricky. Dec 2005.

[11] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast userlevel locking in linux. https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf, 2002.

[12] J. Liedtke and H. Wenske. Lazy process switching. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 15–18, May 2001.

[13] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116–123, 1990.

[14] Lui Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, 1990.

[15] Alexander Züpke. Deterministic fast user space synchronization. In *In Proceedings of the Ninth Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'13*, pages 56–60, 2013.

## APPENDIX A
## PCP-DU EXECUTION FLOW

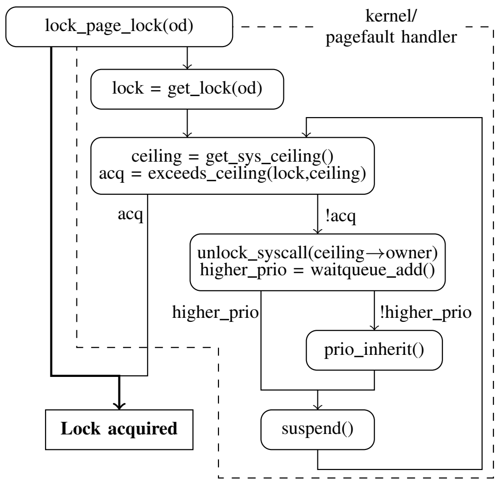

Fig. 5. PCP-DU `lock()` execution flow

Figure 5 shows the execution flow for the `lock()` operation. Here `lock_page_lock()` is the user-space routine that attempts to acquire a lock by writing to the `locked` bitmap of the lock page. When failing to write to the bitmap the kernel will add the current job to the priority wait-queue and suspend until the ceiling is low enough for the current job to obtain the job.

We have implemented two different mechanisms for `lock_page_lock()`. In PCP-DU-PF, when it was predetermined that lock acquisition is not allowed, the lock page is marked read-only. `lock_page_lock()` tries to acquire a lock by seting a bit in the bitmap, resulting in a write fault on failure. In this case the operating systems page-fault handler will be invoked, which is extended to execute the slow locking path. In PCP-DU-BOOL, `lock_page_lock()` uses an atomic compare-and-exchange operation to set the lock bit under the condition that the higest bit in the bitmap is set. When this fails, the routine will invoke the kernel to execute the slow path.

In the slow path, the job repeatedly suspends itself until the ceiling was lowered sufficiently to obtain the lock. On suspension, the `unlock_syscall()` routine increments the `unlock_syscall` counter in the lock page of the system ceiling owner, informing it that another job must be woken up on unlocking. It then sets up a new priority inheritance if the job is the highest priority waiting job in the system and suspends itself.

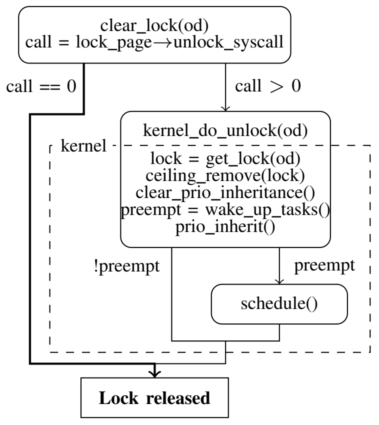

Fig. 6. PCP-DU `unlock()` execution flow

Figure 6 shows the flow for an unlock operation. This demonstrates how the `unlock_syscall` integer is used to determine whether the kernel should be invoked or not. `wake_up_tasks()` will decrement this integer by 1 for every task woken up, and if one or more higher priority tasks are woken up, the scheduler will be invoked to preempt the current job.

## APPENDIX B
## MPCP-DU AND FMLP$^+$-DU EXECUTION FLOW

Figure 7 shows the execution flow for the lock operation. `uLock` is the integer shared between the different tasks. After setting the right bit in `boost`, `uLock` is atomically incremented. The old value returned by this operation is used to determine whether the job successfully acquired the lock or rather needs to invoke the kernel slow path.

On the slow path, the free flag is used to determine whether the job still needs to suspend. If the flag is not set the job adds itself to the wait queue and suspends, otherwise it simply unsets the flag. For MPCP-DU, this wait queue is priority ordered, for FMLP$^+$-DU a FIFO-ordered wait queue is used.

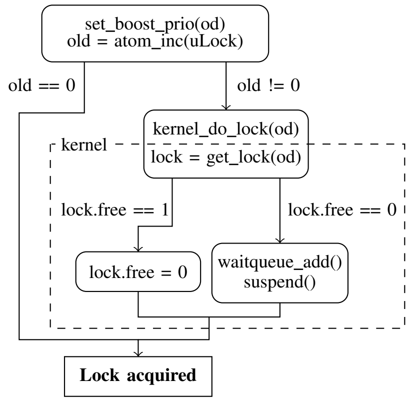

Fig. 7. Multiprocessor `lock()` execution flow

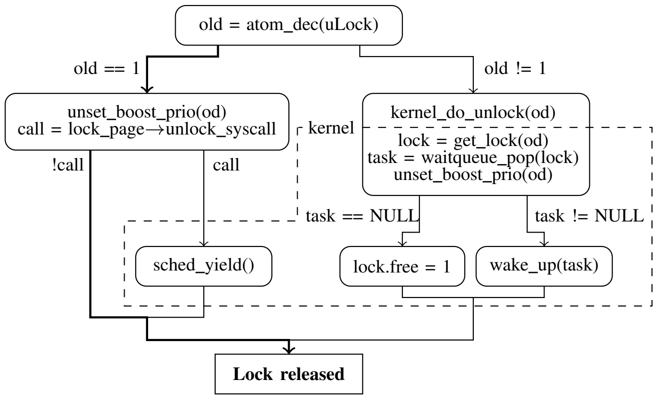

Fig. 8. Multiprocessor `unlock()` execution flow

The unlock execution flow is shown in Figure 8. Atomic decrement on the `uLock` variable determines whether the kernel should be called to wake up the next job in the wait queue, possibly resulting in preemption.

When a higher priority job is on the ready queue but its priority did not exceed the current job's boosted priority, unlock of a mutex should lead to preemption even if no job is blocking on this mutex. For an unlock through the kernel, preemption is thus performed after waking up the next job on the wait queue. For unlock operations that remain in userspace, the kernel determines in advance whether there is such a job on the local ready-queue. If so, it sets the unlock_syscall value in the lock page to inform the current job to invoke the scheduler upon unlocking.

## Appendix C
## Contended lock operation distributions

The distribution for suspending uniprocessor unlock operations is shown in Figure 9. Here the top code paths correspond with the situation where more than one suspended task should be woken up. We observe an overhead of several hundred cycles for the PCP-DU-BOOL and the PCP-DU-PF, both sharing the same unlock code path. We note that the start and end point of the two curves are equal, but the distribution differs This overhead is incurred by userspace code and race condition checks required to prevent the lock state from corrupting when the scheduler interrupted between deciding to unlock through the slow path and reaching the kernel.

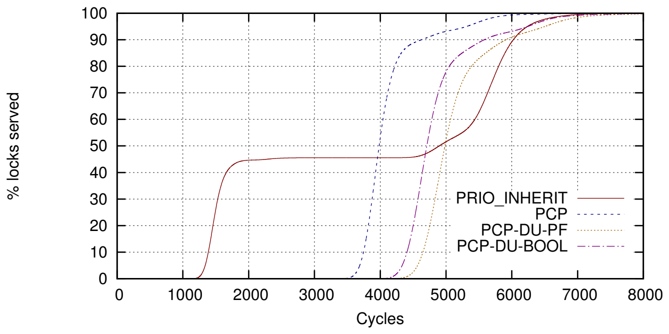

Fig. 9. Cumulative contended unlock time (uniprocessor)

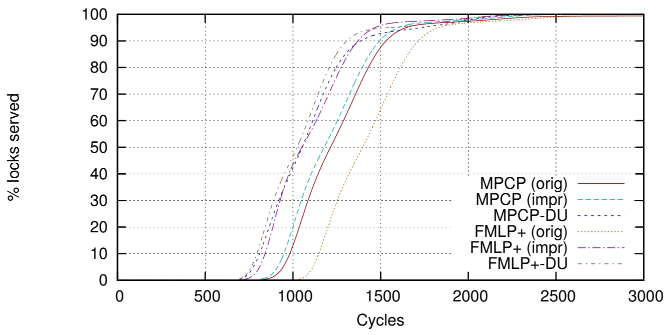

Fig. 10. Cumulative contended lock time (multiprocessor)

For multiprocessor locking protocols, the lock overhead is limited to the user-space code overhead. In graph 10 we can observe that the MPCP-DU and the FMLP$^+$-DU are slightly more efficient than the previous implementations. This is mostly caused by moving the priority boost code to the scheduler. Furthermore, we can observe that using a logical timestamp for priority boosting in FMLP$^+$ improved its contended lock time.