The Case for Migratory Priority Inheritance in Linux: Bounded Priority Inversions on Multiprocessors

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)
Campus E 1 5, D-66123 Saarbrücken, Germany
bbb@mpi-sws.org

Andrea Bastoni
SYSGO AG
Am Pfaffenstein 14, D-55270 Klein-Winternheim, Germany
abastoni@sysgo.com

Abstract

Linux's real-time performance crucially depends on priority inheritance because—*on uniprocessors*—it limits the maximum duration of priority inversion to one critical section per lock while ensuring that tasks remain fully preemptable even when holding locks. Unfortunately, priority inheritance is ineffective *on multiprocessors* under non-global scheduling (*i.e.*, if some tasks may not execute on every processor) in the sense that it does not prevent unbounded priority inversion in all cases. In fact, as shown in the paper, in a multiprocessor system *with* priority inheritance, it is possible for a task to suffer from priority inversion as long as in a uniprocessor system *without* priority inheritance. There is thus currently no predictable locking mechanism suitable for non-global scheduling available in Linux, short of resorting to non-preemptive sections or other latency-increasing mechanisms such as priority boosting. As multicore platforms are becoming more widespread in the embedded domain, this "predictability gap" on multiprocessors limits Linux's viability as a real-time platform.

In this position paper, it is argued that a simple tweak to the priority inheritance algorithm will restore its efficacy on multiprocessors, without breaking POSIX compliance, increasing scheduling latency, or requiring large changes to Linux's scheduling and locking code. In a nutshell, under the proposed *migratory priority inheritance* mechanism, inheritance is applied not only to scheduling priorities, but also to processor affinity masks. The need for migratory priority inheritance and its benefits are explained in detail with a sequence of simple examples. Additionally, a prototype implementation within the Linux kernel is described and potential challenges and simplifications are discussed.

1 Introduction

The PREEMPT_RT patch has been instrumental in turning Linux into a viable platform for demanding real-time applications. Crucially, the PREEMPT_RT patch enables *priority inheritance*[1] [39, 41] for virtually all in-kernel locks, which ensures that real-time tasks are never delayed due to locks accessed only by lower-priority tasks. As Linux consists of many complex subsystems (of which real-time tasks likely access only few), this isolation among real-time and non-real-time tasks is essential to providing suitably short response times in practice. More generally, the combination of fixed-priority scheduling and priority inheritance, as mandated by the POSIX real-time standard [2, 3], has seen tremendous success in uniprocessor real-time systems because it is both analytically sound and robust with regard to real-world engineering concerns. For example, priority inheritance does not require the programmer to specify any additional parameters, it works even if it is not known beforehand which locks a real-time task is going to access, and it transparently allows real-time tasks to share locks with non-real-time tasks (which, in Linux, is unavoidable in kernel space).

Unfortunately, as we explain in detail in §3, classic priority inheritance breaks down in certain scenarios in the multiprocessor case. In particular, the desirable property

[1] See §2 for a review of key concepts.

that the duration of priority inversion (reviewed in §2.2) depends only on critical section lengths is not maintained in all scenarios, even when using priority inheritance. Analytically speaking, this renders priority inheritance ineffective and unsound; practically speaking, this violates the intuition and "best practices" that real-time application developers have acquired over the last two decades.

While the multiprocessor priority inversion problem has received considerable academic attention (see §6), most solutions proposed in the literature make assumptions that are incompatible with the Linux kernel. For instance, many require "priority boosting" of critical sections (reviewed in §3.3), which exposes real-time tasks to delays from *any* critical section in the kernel.

In this position paper, we argue that a simple tweak to Linux's existing priority inheritance mechanism will restore both the analytical properties of priority inheritance and developer expectations, without breaking POSIX compliance or requiring large changes to Linux's scheduling and locking code. The key idea is that inheritance should apply not only to scheduling priorities, but also to the eligibility on which processors a task may execute, which we refer to as "migratory priority inheritance."

Migratory priority inheritance is a straightforward generalization of Linux's current locking semantics (it reduces to priority inheritance on uniprocessors) with the added benefit that it prevents unbounded priority inversions on multiprocessors. Further, it ensures that tasks are never delayed by locks accessed only by lower-priority tasks, both analytically speaking and from the point of view of a developer's intuition (which, for Linux, is arguably as important as analytical correctness).

In the following, we define migratory priority inheritance and argue its benefits (§4) after first demonstrating the shortcomings of both classic priority inheritance (in a multiprocessor context) and priority boosting (§3). Addressing practical implementation concerns, we outline how migratory priority inheritance fits into Linux's existing inheritance implementation (§5). Finally, we relate migratory priority inheritance to prior work on real-time locking (§6). To begin with, we review the uniprocessor case, which is already well-supported by Linux today.

2 Linux on Uniprocessors

Before explaining the shortcomings of priority inheritance on multiprocessors, we need to establish the required real-time scheduling and analysis background. In this section, we briefly review uniprocessor scheduling fundamentals (§2.1), give an example of "unbounded priority inversions" (§2.2), and illustrate priority inheritance (§2.3). Readers familiar with these concepts may safely skip ahead to §3.

2.1 Fixed-Priority Scheduling

Linux implements *fixed-priority* scheduling, where each task is given a fixed numerical priority and tasks are scheduled in order of decreasing priority (in Linux, priorities range from 99 to 1, with 99 being the highest priority). If a task becomes available for execution (*e.g.*, if new input becomes available) while a lower-priority task is scheduled, the lower-priority task is preempted immediately and the higher-priority task is scheduled instead. Fixed-priority scheduling is a predictable real-time scheduling policy and thus amenable to *a priori* analysis. In particular, given a suitable model of the workload, it allows bounding the *worst-case response time* for each task (*i.e.*, the maximum time required by the system to react to a particular input).

Sporadic tasks. The classic model of event-driven, recurrent real-time execution, and the one assumed in this paper, is the *sporadic task model* [34, 36], in which the execution of real-time tasks is modeled as a sequence of jobs. Each real-time task T_i is characterized by three parameters: a *worst-case execution time* (WCET) e_i, a *minimum inter-arrival time* p_i, and a *relative deadline* d_i. Each time that a task T_i is triggered (*e.g.*, when new sensor input becomes available), it *releases* a *job*, which must *complete* within d_i time units. The WCET e_i specifies the maximum amount of processor time required by one job of T_i to complete (*i.e.*, a job of T_i must be scheduled for at most e_i time units within a window of d_i time units after its release). Finally, the p_i parameter specifies the minimum separation between any two job releases of T_i. For historical reasons, p_i is also called the *period* of T_i.

In Linux, a sporadic task can be easily implemented as an infinite loop in which the first statement of the loop body causes the process to wait for an asynchronous event (*e.g.*, expiry of a timer, arrival of a network packet or sensor input, *etc.*). In such an implementation, each iteration of the main loop corresponds to one job, that is, a job is released when the task resumes from the initial wait statement, and completes when the task reaches the first statement again to wait for the next event.

A job is said to be *pending* from its release until it completes. While pending, a job is *ready* when the task is available for scheduling and *suspended* otherwise. A task's *maximum response time* r_i is the maximum time that any of its jobs remains pending. To be deemed correct, the system must ensure that $r_i \leq d_i$ for each task T_i. To this end, *response-time analysis* [6, 29] is used to bound the maximum response time of each task, and thus to verify the temporal correctness of a system *a priori*.

Priority inversion. Response-time analysis is based on the fact that lower-priority tasks normally do not delay higher-priority tasks. Importantly, response-time analysis assumes that a pending job is *not* scheduled (*i.e.*, its completion delayed) only if a higher-priority job is execut-

Task	WCET	Period	Deadline	Critical Section	Priority
T_A	6	20	7		99
T_B	11	20	20	2	98
T_C	6	200	70		97
T_D	11	200	200	2	96

Table 1: Example task set consisting of four tasks (T_A–T_D). The task set is feasible on a uniprocessor under fixed-priority scheduling if priorities are assigned in order of increasing relative deadlines (*i.e.*, with deadline-monotonic priorities [33]) and if priority inheritance is employed.

ing or if the job self-suspended due to locking-unrelated reasons (*e.g.*, due to I/O). A *priority inversion* exists whenever this assumption is violated. That is, any delays not attributable to the scheduling of higher-priority tasks or self-suspensions—if a task *should be* but *is not* scheduled and the processor is idle or a lower-priority task is scheduled instead—are called priority inversions or *blocking*. To determine safe upper bounds on worst-case response times, it is essential to account for all possible blocking: the maximum total duration of priority inversion must be bounded and added to a task's response-time bound.

In the following, we briefly illustrate the danger of unbounded priority inversions, and how to control them using priority inheritance.

2.2 Uniprocessor Priority Inversion

Consider a set of four sporadic control tasks T_A–T_D, of which two have a maximum operating frequency of 50Hz (*i.e.*, a minimum inter-arrival time of 20ms) and two have a maximum operating frequency of 5Hz (*i.e.*, a minimum inter-arrival time of 200ms), as given in Table 1. Note that tasks T_A and T_C have *constrained deadlines* to ensure their timely completion (*i.e.*, their relative deadline is shorter than their minimum inter-arrival time), and that tasks T_B and T_D share a resource (*i.e.*, have shared state) with an associated maximum critical section length of two.

The given task set is feasible on a uniprocessor using fixed-priority scheduling with *deadline-monotonic* (DM) priorities [33]. That is, the task set can be scheduled such that all deadlines are met if priorities are assigned in order of increasing relative deadlines (*e.g.*, task T_A is assigned the highest priority, task T_B is assigned the second-highest priority, *etc.*). If all tasks were *independent* (*i.e.*, if no tasks were to lock shared resources), or if no contention arises, then all deadlines are indeed met. However, excessive blocking can arise if locks become contended.

Suppose that access to the shared resource is controlled using a regular, real-time-unaware mutex. A possible schedule illustrating the risk of "untimely" job releases is shown in Figure 1. In this and all later examples, only a single job of each sporadic task is shown for the sake of simplicity. In the depicted example schedule, task T_B misses its deadline at time 25 because it is indirectly delayed by the lower-priority task T_C while waiting for task T_D to release the lock. Since T_D has a lower priority than T_C, T_B is indirectly delayed for the duration of the entire execution of T_C's job. This violates the intuition un-

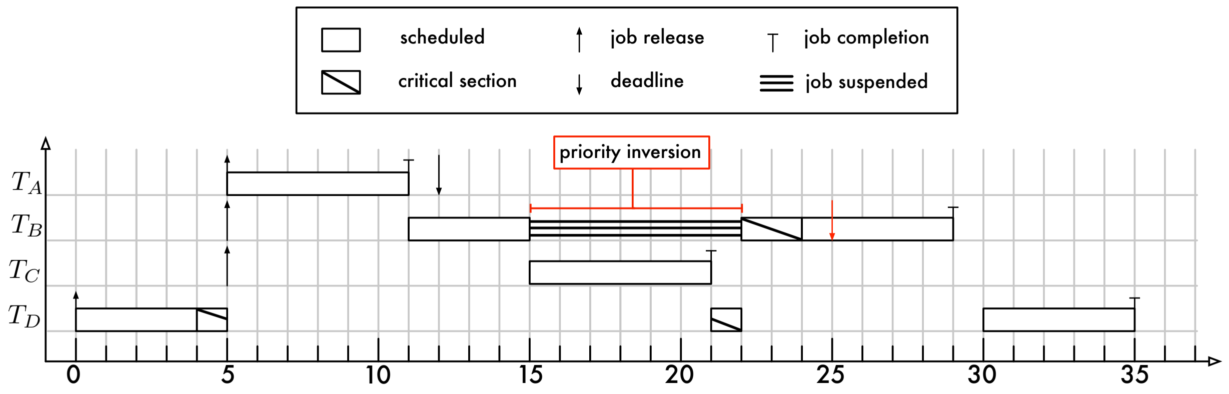

Figure 1: Example of an extended priority inversion on a uniprocessor in the absence of priority inheritance. Task T_C (priority 97) delays the completion of the lock-holding task T_D (priority 96), which delays the higher-priority task T_B. A long priority inversion and a deadline miss at time 25 results. The depicted legend applies to Figure 2 as well.

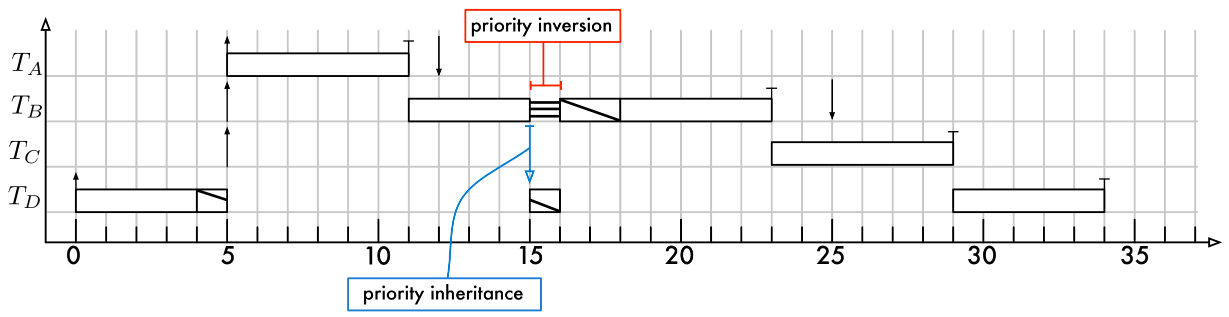

Figure 2: Example of priority inheritance on a uniprocessor. The lock-holding task T_D is not preempted by task T_C at time 15 since it inherits the priority of task T_B at the time. As a result, T_B is able to meet its deadline at time 25 since it is only briefly blocked. See Figure 2 for a legend.

derlying fixed-priority scheduling—a lower-priority task should never delay a higher-priority task—and hence constitutes a priority inversion. In total, T_B incurs seven time units of blocking in this example (from time 15 until time 22). As a result of the unexpected delay, the job misses its deadline at time 25.

2.3 Classic Priority Inheritance

A classic way to control this well-known problem in uniprocessor real-time systems is to employ *priority inheritance* [39, 41], where the priority of a lock-holding task is temporarily raised to the maximum of its own priority and that of any task that it blocks. While some amount of priority inversion is fundamentally unavoidable when using locks, priority inheritance ensures that the maximum blocking due to each lock is bounded by the associated maximum critical section length.

The benefit of bounding priority inversions to critical section lengths is demonstrated by the example schedule depicted in Figure 2. In the example, task T_C does not preempt task T_D at time 15 since T_D inherits the higher priority of task T_B while T_D blocks T_B. Task T_D can thus quickly finish its critical section (which causes it to lose the benefit of priority inheritance) and ceases to block task T_B at time 16. As a result, all deadlines are met in the depicted example, and, in fact, for any possible arrival sequence of the considered task set.

With priority inheritance, priority inversions are considered *bounded* because their length depends only on the maximum critical section length. In contrast, without priority inheritance, priority inversions are potentially *unbounded* since the maximum duration of priority inversion then depends on entire job execution costs (recall Figure 1), which are typically much larger than critical section lengths (*i.e.*, locks are typically only held for a small fraction of a task's execution).

This completes our review of uniprocessor real-time scheduling fundamentals. To summarize, *on uniprocessors*, priority inheritance limits locking-related priority inversion to one critical section (per accessed lock). Since even otherwise independent tasks frequently contend for locks when executing in kernel space, it is not surprising that priority inheritance is in widespread use, and in fact mandated by the POSIX standard [2, 3]. In Linux, priority inheritance has been available since version 2.6.18 [19] and is enabled by default for most kernel locks by the PREEMPT_RT patch. Linux is thus well-equipped for hosting real-time workloads *on uniprocessors*. Unfortunately, *on multiprocessors*, classic priority inheritance does not bound priority inversions in all cases, which we demonstrate with an example next.

3 The Predictability Gap

To motivate the need for multiprocessors, suppose that the example system's specification is updated to incorporate support for more-demanding operating environments. Specifically, consider the case that tasks T_C and T_D are now required to also support operating frequencies as high as 50Hz (*i.e.*, the periods of tasks T_C and T_D are shortened to 20ms), with their relative deadlines scaled accordingly. The resulting task set is summarized in Table 2.

Note that only the temporal *constraints* have been tightened to reflect the increased maximum rate of events, which does not affect the actual implementation—the WCET parameters thus remain unchanged. The updated task set is therefore no longer feasible on a uniprocessor. That is, there does not exist a schedule such that all deadlines are met if all tasks simultaneously exhibit worst-case behavior. Assuming that it is not possible to use a faster uniprocessor (*e.g.*, due to thermal, energy, or budget constraints), the only remaining option is to deploy the task set on a multiprocessor consisting of at least two cores.

Task	WCET	Period	Rel. Deadline	Critical Section	Priority	Processor
T_A	6	20	7		99	1
T_B	11	20	20	2	97	1
T_C	6	20	7		98	2
T_D	11	20	20	2	96	2

Table 2: Example task set. The task set is feasible on two processors under P-FP scheduling if priorities are assigned in order of non-decreasing relative deadlines (*i.e.*, with deadline-monotonic priorities). Further, neither tasks T_A and T_C nor tasks T_B and T_D may be assigned to the same processor.

3.1 Multiprocessor Scheduling

In the following, let us consider the case of *partitioned fixed-priority* (P-FP) scheduling, where each task is statically assigned to one of the cores, on a dual-core processor. From a scheduling point of view, partitioned scheduling reduces the multiprocessor system to a collection of uniprocessors, albeit with locking dependencies among the cores. Besides the reduction to uniprocessor scheduling, partitioned scheduling has many advantages in practice (maximal cache affinity chiefly among them) and is thus in widespread use. In Linux, partitioned scheduling corresponds to assigning each task a *processor affinity mask* with only a single bit set.[2]

Diametrically opposed to partitioned scheduling is *global scheduling*, where each task may execute on each processor. As we will revisit later, global scheduling is the only multiprocessor real-time scheduling variant under which classic uniprocessor priority inheritance works as expected and reliably limits maximum blocking. However, global scheduling is generally subject to higher overheads and, at least anecdotally, less commonly used in practice. In the remainder of this paper, we restrict our focus to P-FP scheduling, which is inarguably relevant to Linux since it is officially supported by Linux (by means of the processor affinity mask API).

As an aside, it is of course also possible to configure hybrid scheduling policies that combine aspects of both partitioned and global scheduling. For example, under *clustered scheduling* [8, 18], the set of processors is divided into non-overlapping subsets (or *clusters*) and each task is assigned to one of the clusters. For the purpose of this paper, all non-global scheduling policies (including partitioned and clustered scheduling) are alike in the sense that classic priority inheritance is not sufficient in all cases; we thus focus on the simpler P-FP case.

Processor assignment. Assuming P-FP scheduling, let us first consider the assignment of tasks to processors. Since both task T_A and task T_C have only little slack (*i.e.*, the difference between the relative deadline and the worst-case execution cost is only one millisecond), it is clear that the two tasks must be assigned to different processors. Similarly, tasks T_B and T_D must also be assigned to different processors since each requires up to eleven milliseconds of processor service within a scheduling window of 20 milliseconds. As a result, the only feasible way to partition the task set is to create two symmetric partitions, consisting of tasks T_A and T_B and tasks T_C and T_D, respectively (swapping tasks T_A and T_C or tasks T_B and T_D does not substantially change the assignment since each pair of tasks has identical parameters).

Priority assignment. Next, consider the choice of priorities. When comparing the slack of tasks T_A and T_C with the WCET of tasks T_B and T_D, it is obvious that T_A and T_C necessarily require higher priorities than T_B and T_D. Since T_A and T_C reside on different processors, the highest and second-highest priority may be assigned to either one; the same applies to T_B and T_D with regard to the remaining priorities.

The two preceding observations lead to the processor and priority assignment summarized in Table 2: tasks T_A and T_B are assigned to processor 1 (indicated by white "boxes" in Figures 3–7) and tasks T_C and T_D assigned to processor 2 (indicated by gray "boxes" in Figures 3–7). Further, task T_A has the highest priority, task T_C has the second-highest priority, task T_B has the third-highest priority, and task T_D has the lowest priority.

It is important to realize that the processor assignment and scheduling priorities were not chosen arbitrarily. Rather, all other priority and processors assignments are *infeasible*, or identical apart from renaming. In contrast, the priority and processor assignment given in Table 2 is feasible, that is, it is *possible* to schedule the task set such that all deadlines are always met. However, this is not the case when employing classic priority inheritance, as is illustrated next.

[2] A processor affinity mask specifies the set of processors that a task may be scheduled on (represented as a bit string). Processor affinity masks are not part of the POSIX standard, but are supported in various forms by most UNIX-like real-time operating systems. In Linux, a task's processor affinity mask is specified using the `sched_setaffinity(2)` system call.

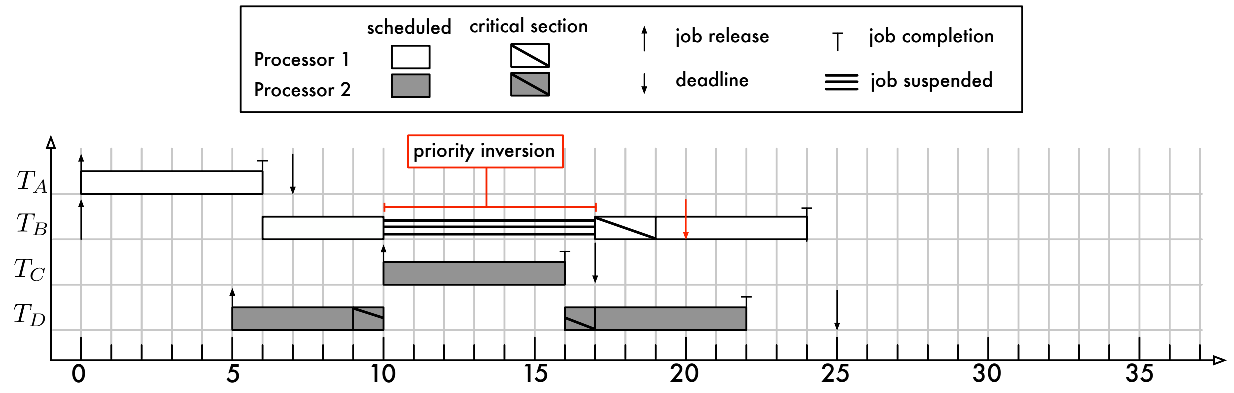

Figure 3: Example of an extended priority inversion under P-FP scheduling despite priority inheritance. Task T_D is not scheduled despite inheriting T_B's priority since a local task, T_C, has still higher priority. Nonetheless, task T_B suffers a priority inversion while waiting since no higher-priority task is scheduled on its processor. See Table 2 for task parameters. The legend applies to Figures 4–7 as well.

3.2 Priority Inheritance is Ineffective

Consider the example shown in Figure 3, which depicts a schedule of the task set given in Table 2 assuming classic priority inheritance is applied across processors (as it is in Linux today). Note that task T_B misses a deadline at time 20, despite priority inheritance, and despite the fact that the total processing capacity was doubled to accommodate the increased demand. This failure arises because task T_B is blocked from time 10 until time 17 while waiting for the lock that task T_D acquired at time 9. Task T_B incurs such excessive blocking *despite priority inheritance* because its assigned priority is lower than that of task T_C (and necessarily so; if task T_B's priority were higher than that of task T_C, then task T_C could miss a deadline instead). Task T_B's delay constitutes a priority inversion since no higher-priority task is scheduled on task T_B's processor while it waits (*i.e.*, from the point of view of response-time analysis, task T_B should be scheduled, but it is not). In fact, task T_B's delay even constitutes an unbounded priority inversion because its length depends on the WCET of T_C (and not just on the critical section length).

Further, when comparing Figure 3 with Figure 1, it is apparent that, in the multiprocessor schedule *with* priority inheritance, task T_B incurs as much blocking as in the uniprocessor example *without* priority inheritance! The purpose of a real-time locking protocol is to ensure bounded blocking in *all* cases, but as demonstrated above, classic priority inheritance under P-FP scheduling fails to bound priority inversions in some cases: *classic priority inheritance is ineffective under partitioned scheduling.*

Recall that the priority and processor assignment given in Table 2 was not chosen arbitrarily; rather, it is the only feasible choice for the example task set (apart from renaming, which does not alter the resulting schedule). Yet it is not possible to correctly schedule it in all cases using priority inheritance. This demonstrates that classic priority inheritance—as currently implemented in Linux, and made default by the PREEMPT_RT patch—is not the best choice for partitioned scheduling.

3.3 The Classic Solution: Priority Boosting

It has long been known that priority inheritance alone is not sufficient to ensure bounded blocking in all cases, and several alternative real-time locking protocols for partitioned scheduling have been proposed in the literature (see §6 for a review). Common to all of the proposed protocols is that they avoid unbounded priority inversions with *priority boosting* [38–40]. Under priority boosting, the priority of a lock-holding task is unconditionally raised above that of other, non-lock-holding tasks. As a result, it is not possible for tasks to be preempted by newly-released jobs, which in turn ensures that locks are released quickly.

Priority boosting is illustrated in Figure 4, which shows a schedule of the same scenario as in Figure 3, but this time assuming that tasks are priority-boosted while holding locks. Note that task T_C cannot preempt task T_D at time 10 even though T_C normally has higher priority. Since task T_D is holding a lock, it benefits from priority boosting and thus remains scheduled until it finishes its critical section at time 11. As a result, task T_B is blocked only briefly and completes before its deadline.

3.4 Increased Scheduling Latency

Given that the earliest shared-memory multiprocessor locking protocol based on priority boosting—namely, the classic *multiprocessor priority-ceiling protocol* (MPCP) [38, 39]—was proposed more than 20 years ago, why have such protocols not yet caught on as the *de facto* "default"

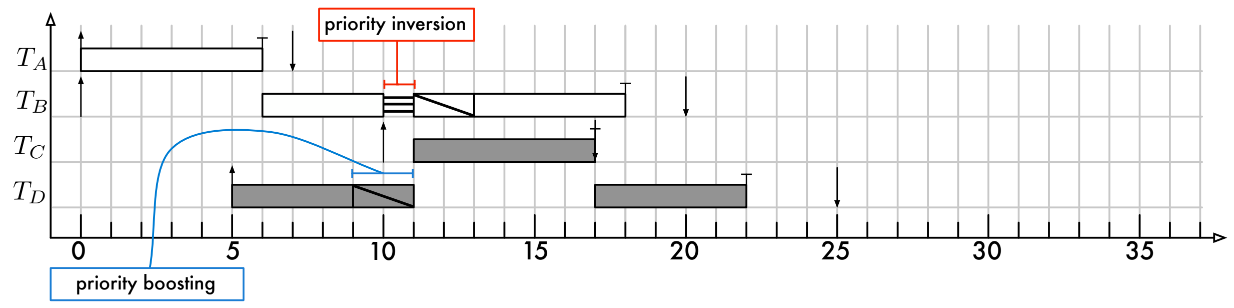

Figure 4: Example of priority boosting under P-FP scheduling. Note that of task T_D's critical section is not interrupted by the activation of task T_C at time 10 since T_D's priority is elevated while holding a lock. See Figure 3 for a legend.

choice? We believe the reason is that priority boosting is a rather "disruptive" progress mechanism, as *unconditionally* expediting the completion of lock-holding jobs comes at the expense of delaying other, possibly much more urgent jobs. This indiscriminate delaying of higher-priority jobs can result in increased scheduling latencies and response times, thus significantly reducing the real-time capabilities of the system.

The potential for deadline misses due to priority boosting is illustrated in the example schedule shown in Figure 5, which shows a schedule resulting from a problematic arrival sequence of the task set previously shown in Figure 4. In the depicted example, the two higher-priority tasks with short deadlines, T_A and T_B, arrive just after task T_D has acquired a lock. Task T_C on processor 2 arrives shortly after T_D became priority boosted at time 4 and thus is not scheduled until time 6, when the normal priority of task T_D is restored. This increase in scheduling latency exceeds T_C's slack; a deadline miss thus results. Task T_A on processor 1 also misses its deadline at time 12, but for a slightly different reason. It arrives while T_B is suspended and thus is scheduled right away. However, when T_B resumes at time 6, it is now holding a lock and thus benefits from priority boosting. Hence it preempts T_A (the task with the highest unboosted priority) and delays it by two milliseconds. Here, T_A incurs a priority inversion even though it does not require a lock.

This side effect of priority boosting makes it problematic for the Linux kernel: since priority boosting affects *all* locks (or at least all locks accessed by real-time tasks), it exposes real-time tasks to delays from potentially *any* critical section within the Linux kernel (or possibly even user-space tasks). This is only a viable option if all critical sections in any part of the kernel are known with certainty to be so short as to not cause significant delays. Given Linux's large, complex, and rapidly changing code base, this is not a realistic assumption to make. Instead, we argue that a simple tweak to priority inheritance could restore its efficacy under partitioned scheduling without introducing any of the downsides of priority boosting.

4 The Case for Migratory Priority Inheritance

It is easy to see why priority boosting is problematic—priority boosting essentially turns critical sections into non-preemptive sections—but why exactly is priority inheritance ineffective? As we explain in the following, the root cause is that it is meaningless to compare priorities across partitions (from an analytical point of view).

On a uniprocessor, the key guarantee provided by classic priority inheritance is the following: if a priority inversion exists, then (one of) the lock-holding task(s) directly or indirectly responsible for the blocking is scheduled (assuming the absence of deadlock and that lock-holders do not suspend for locking-unrelated reasons). This is because a priority inversion exists only if a waiting task has a higher priority than all currently ready tasks; due to (transitive) priority inheritance, the blocked task's priority is made available to a lock holder, which is consequently scheduled without delay. In other words, priority inheritance establishes an analytical link between processor availability and priority inversion.

On a non-globally-scheduled multiprocessor, this property breaks down: it is possible for a lock holder to be preempted even though a blocked task is incurring a priority inversion at the same time (recall Figure 3). There is thus no link between processor availability and priority inversion, which renders it ineffective from the point of view of worst-case analysis. The reason for this disconnect is that the processor on which the priority inversion is incurred (*i.e.*, the processor that would be available) may not be the one on which the lock holder resides. That is, on a multiprocessor, a priority inversion still signals the availability of a processor (with respect to the blocked task's priority level); however, if the lock holder may not migrate to the available processor(s), then an unbounded priority inversion may still arise.

As an aside, this dependence on task migration explains why classic priority inheritance *is* effective under

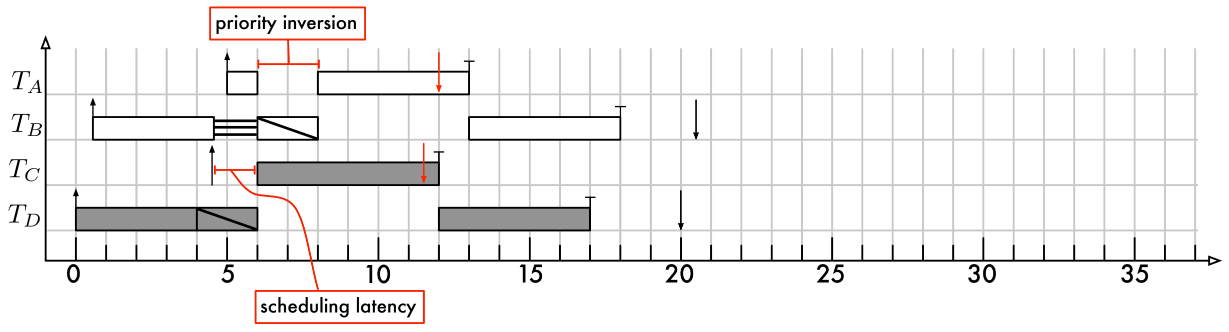

Figure 5: Example of increased scheduling latency and blocking due to priority boosting under P-FP scheduling. Task T_C misses its deadline because the processor is occupied by a priority-boosted task when its job is released, which causes it to suffer increased scheduling latency. See Figure 3 for a legend.

global scheduling—under global scheduling, lock holders may always migrate to all processors, which restores the link between processor availability and priority inversion.

4.1 A Simple Solution

The above considerations show that task migration is necessary to some extent if unbounded priority inversions and priority boosting are both to be avoided. However, short of resorting to global scheduling, what can be done to restore the efficacy of priority inheritance on multiprocessors?

We propose the following solution: augment priority inheritance with *processor affinity mask inheritance*. That is, inheritance should not only pertain to scheduling priority, but should also extend to the *eligibility* to execute on a particular processor. For brevity, we refer to the combination of priority inheritance and processor affinity mask inheritance as *migratory priority inheritance*.

Migratory priority inheritance works as follows. Let π_x denote the assigned priority of a given task T_x, and let M_x denote the processor affinity mask of T_x (*i.e.*, M_x is the set of processors that T_x has been assigned to). If T_x is blocked by another task T_y, then T_y inherits T_x's *eligibility tuple* (π_x, M_x), with the interpretation that T_y may execute on any of the processors included in M_x with priority π_x. The inherited eligibility tuple takes effect in addition to all other inherited eligibility tuples (if any) and T_y's own eligibility tuple (*i.e.*, T_y may always execute with priority at least π_y on any of the processors in M_y).

To be precise, the effective scheduling parameters of a task T_y are determined as follows. Let E_y denote the set of eligibility tuples currently inherited by T_y (if any, including all tuples available due to transitive inheritance). Then T_y's *effective processor affinity mask* M'_y is the set of processors

$$M'_y \triangleq \bigcup \{M_i \mid (\pi_i, M_i) \in E_y \cup (\pi_y, M_y)\}.$$

Paraphrased, under migratory priority inheritance, a task may execute on any processor that it or one of the tasks that it blocks is eligible to execute on.

Further, on each processor $p \in M'_y$, the *effective scheduling priority* $\pi'_{y,p}$ of T_y is defined as follows:

$$\pi'_{y,p} \triangleq \max\{\pi_i \mid (\pi_i, M_i) \in E_y \cup (\pi_y, M_y) \\ \wedge p \in M_i\}.$$

In other words, priority inheritance is applied on a per-processor basis such that a task may have a different priority on each processor that it is eligible to execute on.

At any time and on each processor p, the task eligible on p with the highest effective priority on processor p is scheduled unless it is already scheduled elsewhere (*i.e.*, Linux's usual push/pull semantics are applied with regard to effective priorities and processor affinity masks). While the focus in this paper is P-FP scheduling (where each M_x contains only a single processor), the presented mechanism seamlessly works with arbitrary processor affinity masks, and reduces to classic priority inheritance on uniprocessors and under global scheduling.

Most importantly, it can be shown that migratory priority inheritance is analytically sound and predictable: it limits priority inversions in all cases, which we illustrate with examples next, and can thus be used to determine bounds on worst-case blocking *a priori*.

4.2 Bounded Priority Inversion

The schedule in Figure 6 demonstrates that migratory priority inheritance is effective at bounding priority inversion without inducing blocking in higher-priority tasks. The schedule shows the same arrival sequence as in Figure 3 (which assumes classic priority inheritance), but assumes that migratory priority inheritance is used instead. Note that, at time 10, the lock-holding task T_D is immediately preempted by the arrival of the higher-priority task T_C, thereby shielding task T_C from incurring any additional

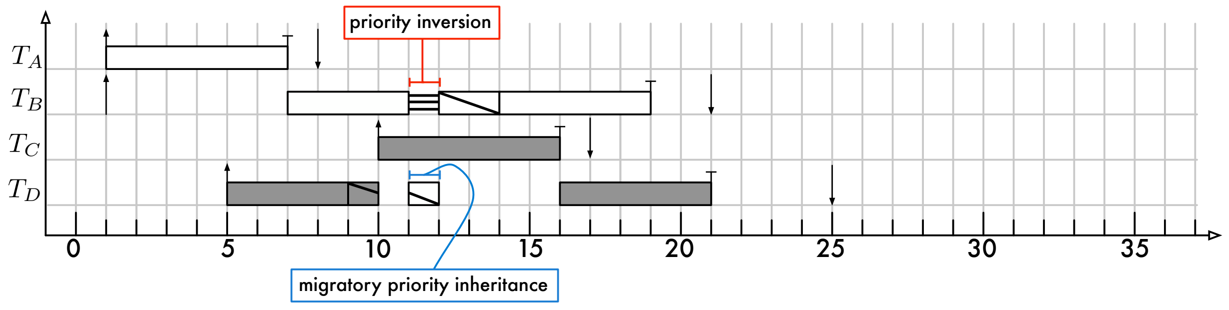

Figure 6: Example of migratory priority inheritance under P-FP scheduling. When task T_B becomes blocked on task T_D at time 11, T_D inherits T_B's processor affinity mask and thus is able to migrate to processor 1 to complete its critical section without further delay. See Figure 3 for a legend.

delays due to unrelated critical sections. As a result, task T_C completes in time.

Migratory priority inheritance takes effect when task T_B becomes blocked on task T_D at time 11. Under migratory priority inheritance, T_D is now eligible to execute on processor 1 with priority 97 (task T_B's priority) since task T_D inherits task T_B's processor affinity mask in addition to task T_B's priority, and also on processor 2, but only with priority 96 (task T_D's own priority) since task T_B's priority is only valid on processors included in task T_B's processor affinity mask. Task T_D thus migrates to processor 1 where no higher-priority task is currently ready. This allows T_D to finish its critical section at time 12, which causes it to stop inheriting T_B's processor affinity mask and priority. Task T_D thus becomes ineligible to execute on processor 1 and is migrated back to processor 2 (where the higher-priority task T_C is still executing). Since migratory priority inheritance ensures the timely completion of T_D's critical section, T_B is blocked only briefly from time 11 until time 12 and completes before its deadline.

In general, migratory priority inheritance ensures that the maximum priority inversion (per lock access) is limited to the length of one critical section, thereby offering a progress guarantee similar to priority inheritance on uniprocessors. As in the uniprocessor case, this guarantee derives from the definition of priority inversion. Recall that a suspended task suffers priority inversion only if no higher-priority task is scheduled (with respect to some processor included in its processor affinity mask). This implies that, due to processor affinity mask inheritance, the blocking task is eligible to execute on a processor on which, due to priority inheritance, its effective priority is sufficient to be scheduled without delay.

4.3 Unaffected Worst-Case Latency

A key property of priority inheritance on uniprocessors is that a task is never delayed due locks accessed only by lower-priority tasks. It is this very property that priority boosting lacks and which causes the detrimental increase in worst-case scheduling latency discussed in §3.4. In contrast, this isolation of high-priority tasks from low-priority activity is maintained under migratory priority inheritance.

This is demonstrated by the example depicted in Figure 7, which shows the same arrival sequence as in Figure 5 assuming migratory priority inheritance instead of priority boosting. The benefits of migratory priority inheritance are immediately apparent as no deadlines are missed. In particular, both tasks T_C and T_A incur no delays when they are released at times 4.5 and 5, respectively. Further, note how task T_D progresses in the execution of its critical section whenever at least one of the two processors is available: initially, T_D starts execution on (its assigned) processor 2 at time 4, but when it is preempted by T_C at time 4.5, T_D migrates to processor 1. At time 5 T_D is again preempted since, with the arrival of T_A, there is no longer a priority inversion (T_B is still suspended, but it is no longer the highest-priority task on its processor and thus would not be scheduled even if it were ready). Finally, at time 10.5, T_D migrates back to processor 2 when its assigned processor becomes available.

Overall, this example shows that migratory priority inheritance ensures the rapid completion of critical sections that cause priority inversions without imposing undue delays on independent higher-priority tasks.

4.4 Fast Path and Overheads

An important concern in practice is implementation efficiency in the common case. That is, while worst-case performance is clearly important in real-time systems, many applications hosted on Linux demand both real-time predictability and high throughput. With regard to the latter requirement, it is important for the common, uncontended case to be as efficient as possible.

Common-case efficiency is another weak point of priority boosting since it requires effective priority changes

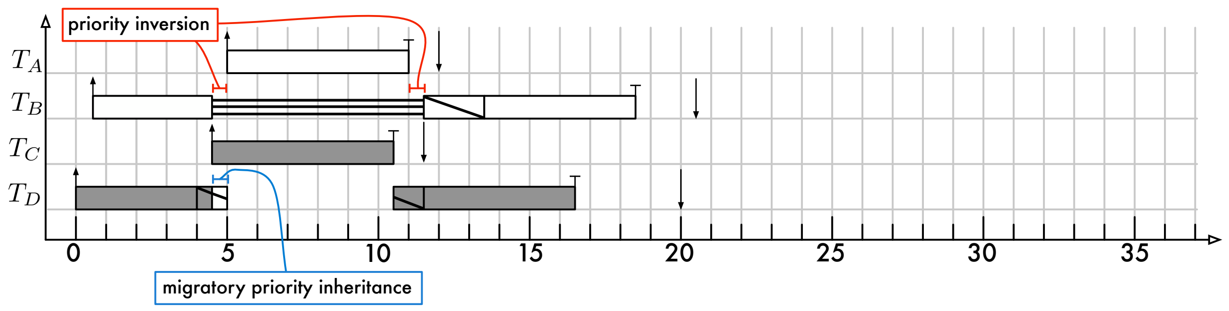

Figure 7: Example of migratory priority inheritance under P-FP scheduling. Since migratory priority inheritance takes effect only while a task suffers a priority inversion, it does not increase the scheduling latency of higher-priority tasks. See Figure 3 for a legend.

for each critical section, which involves system calls in Linux. In contrast, migratory priority inheritance takes effect only when there is contention and does not require any kernel involvement in the common, uncontended case. It is thus fully compatible with Linux's futex interface.[3]

The only overhead increase (compared to regular priority inheritance) is the need to migrate tasks when lock holders are preempted. However, these migrations occur only rarely (only in the contended case, and even then only if an "untimely" preemption of the lock holder occurs) and only if cache affinity is lost anyway (due to the triggering preemption). Further, lock-holder migrations do not involve the full working set of the migrating task; rather, only the working set of the critical section, which is typically minuscule, is accessed on the remote core.

4.5 POSIX and Developer Expectations

In the context of Linux, a another practical concern is compliance with the POSIX standard. In particular, Linux generally strives to satisfy the POSIX "Multi-Purpose Realtime System Profile" (PSE54) [2], and any change to Linux's locking primitives must be compatible with both the letter and, perhaps more importantly, the spirit of the POSIX standard. We believe migratory priority inheritance satisfies both requirements.

First, with regard to the actual standard, compliance in the multiprocessor case is trivial due the simple fact that the POSIX real-time profile does not address multiprocessor issues; any solution that does not conflict with the uniprocessor requirements is thus "compliant." In particular, Linux's processor affinity mask implementation is not (yet) covered by the POSIX standard, even though the concept of processor affinity masks is widespread among UNIX-like real-time operating systems; adding processor affinity mask inheritance is thus unproblematic. Further, migratory priority inheritance reduces to regular priority inheritance on uniprocessors. Hence, it is possible to introduce migratory priority inheritance into the Linux kernel without breaking POSIX compliance.

Second, with regard to the "spirit" of the standard, we believe that migratory priority inheritance restores key properties that application developers have come to expect from priority inheritance on uniprocessors:

1. it only takes effect in the case of contention, thereby avoiding overheads when kernel intervention is not required;
2. high-priority tasks are shielded from any delays due to locks accessed only by lower-priority tasks, which makes it safe to enable migratory priority inheritance for all kernel locks;
3. migratory priority inheritance does not require developers to specify protocol parameters (such as priority ceilings);
4. migratory priority inheritance works with arbitrary processor affinity masks (*i.e.*, it fully supports all scheduling options supported by Linux); and
5. whenever there exists a priority inversion, at least one blocking task is scheduled on some processor (assuming the absence of deadlock and that tasks do not self-suspend for locking-unrelated reasons), thereby ensuring the timely completion of blocking critical sections.

To summarize our argument, classic priority inheritance (when applied to partitioned scheduling) satisfies properties 1–4, but does not guarantee in all cases that priority inversions will be bounded, and priority boosting satisfies properties 3–5, but may unconditionally delay higher-priority tasks. In contrast, migratory priority inheritance satisfies all five properties. In particular, while we

[3] A *fast user-space mutex* (*futex*) [25] is a mutex implementation in which the lock word is shared between kernel and user space. This allows tasks to acquire and release locks without system calls in the uncontended case. Kernel intervention is only required when blocking occurs.

have focused on P-FP scheduling in this paper for ease of exposition, it is important to note that migratory priority inheritance works with arbitrary processor affinity masks, including global scheduling (where it reduces to regular priority inheritance). We believe that migratory priority inheritance can be realized in Linux with acceptable overheads and thus deem it an attractive solution to the "predictability gap" that currently exists under Linux on non-globally-scheduled multiprocessors.

Next, we outline an implementation of migratory priority inheritance in Linux, point out potential challenges, and discuss a simplified variant.

5 Implementation Considerations

Linux offers a well-structured and modular implementation of the classic priority inheritance protocol. In the following, we first review Linux's existing priority inheritance infrastructure and then discuss how migratory priority inheritance can be realized in Linux. Some familiarity with the Linux kernel is assumed. Readers primarily interested in the algorithmic properties of migratory priority inheritance may safely skip this section.

The considerations presented in this section are based on a prototype implementation of migratory priority inheritance in the 3.5 "vanilla" kernel, that is, without the PREEMPT_RT patch, which does not modify the mechanisms relevant to the following discussion. Even though the specific kernel version (which is the most recent at the time of writing) may soon be outdated, the ideas underlying Linux's priority inheritance support have not substantially changed since the original priority inheritance patches [19] and are likely to remain valid in future versions as well.

5.1 The Current Implementation

The abstraction of a lock (or *mutex*) is provided by the `rt_mutex` structure, which encapsulates information about both the task currently owning the mutex and the tasks currently waiting to acquire the mutex (*waiters*). Lock ownership information and the presence of waiters are efficiently encoded together and can be updated with a single atomic compare-and-exchange operation.[4]

A waiting task is represented by a structure named `rt_mutex_waiter`, which is allocated on a task's stack when it fails to acquire a lock. Tasks waiting for a mutex are enqueued in a wait queue, which is sorted by task priority and contained in the lock's `rt_mutex` structure. Multiple tasks can be waiting to acquire the same lock and a single task can hold multiple locks (and thus block tasks waiting for different locks). Under priority inheritance, the priority of a lock-holding task T_x must be raised to the priority of the highest-priority task (the *topmost waiter* in kernel parlance) currently waiting to acquire *any* of the locks currently held by T_x.

Top waiters. Clearly, quickly retrieving the current (inherited) priority of a task T_x is key to implementing priority inheritance efficiently. In Linux, the highest-priority waiter (*i.e.*, the *top waiter*) of each lock owned by T_x is enqueued in a priority-sorted list. This list is commonly referred to as T_x's *top waiters list*. Since the top waiters list is priority-sorted, the head of the list is the topmost waiter, and holds the highest priority among the tasks waiting for all the locks currently owned by T_x. Under classic priority inheritance, the topmost waiter's priority is exactly the priority that should be inherited by T_x.

Priority adjustments. The (inherited) priority of a lock-holding task T_x may have to be updated whenever T_x releases a lock and whenever a higher-priority task joins or leaves the set of tasks waiting for a lock owned by T_x (*e.g.*, due to timeout expiration or signal delivery). When a task T_y fails to acquire a lock L that is currently held by another task T_x, T_y is registered among the waiters for L, and, if T_y is the new top waiter with regard to L, T_x's top waiters list is updated. In this case, the priority of the lock holder T_x is re-evaluated.

The required priority adjustment is performed by the `__rt_mutex_adjust_prio()` function, which invokes `rt_mutex_getprio()` to retrieve the effective (*i.e.*, possibly inherited) priority of a task, and invokes `rt_mutex_setprio()` to update the priority of the task if it differs from the current one. If a task has no waiters, `rt_mutex_getprio()` simply returns the task's assigned base (*i.e.*, not inherited) priority. The function `rt_mutex_setprio()` acquires the appropriate runqueue lock needed to safely update the priority of a task.

5.2 Migratory Priority Inheritance

Given the simplicity of migratory priority inheritance, we believe that it can be supported within the current priority-inheritance Linux framework with a limited amount of changes. Nonetheless, some of these changes could be nontrivial to implement correctly.

The key approaches to reduce the impact on the existing code are:

- appropriately construct a list of relevant highest-priority waiters that increase the effective priority and/or extend the effective processor affinity mask of the lock-holding task;
- promptly update the priority of a lock-holding task

[4] The actual design of `rt_mutex` and the operations needed to update the ownership of a mutex are slightly more involved. An introductory-but-detailed description of the design of real-time mutexes can be found in the kernel documentation.

if its current processor is included in the processor affinity mask associated with some newly inherited, higher priority; and

- update the push/pull migration logic to consider per-processor priorities that may differ among processors for lock-holding tasks.

Next, we discuss the data structures and operations needed to support migratory priority inheritance.

Waiters, top waiters, and top masks. As under normal priority inheritance, migratory priority inheritance should ensure that locks will be acquired in the "right order" by the tasks waiting for them. Specifically, the highest-priority waiter should become the next lock owner when the current owner releases it. Therefore, Linux's current implementation of a priority-sorted queue of waiters is not modified under migratory priority inheritance.

However, the notion of top waiters differs under migratory priority inheritance from the one under normal priority inheritance. Specifically, different waiters may have different processor affinity masks. Therefore, more than one "top waiter" may exist for each lock in the sense that more than one waiting task may contribute to the lock holder's effective priority on some processor. To keep track of these "top waiters," a *top-mask list* is added to each lock. This list contains the eligibility tuples of tasks (*i.e.*, *top-mask waiters*) that have non-redundant processor affinity masks (*i.e.*, that are eligible to execute on some processor that none of the higher-priority tasks blocked on the same lock are eligible to execute on).

Detecting whether a newly blocked tasks contributes a non-redundant processor affinity mask can be easily accomplished if the top-mask list sorted in task priority order. To do so, the list is traversed from highest to lowest priority and the union of the processors already contributed by equal-or-higher-priority waiters is accumulated.

The `squashed_mask_list`. Under migratory priority inheritance, a lock-holding task T_x's effective priority *on each processor* is defined by the eligibility tuples (*i.e.*, the priorities and processor affinity masks) of its top-mask waiters, in addition to its assigned base priority and processor affinity mask. However, for efficiency reasons, it is undesirable to enumerate all top-mask waiters as part of each scheduling decision.

Instead, the required information about a task T_x's top-mask waiters (which are possibly waiting on different locks) can be compactly represented by caching, for each priority level l, where $1 \leq l \leq 99$, the associated union of affinity masks of all top-mask waiters with assigned priority l. This *broadest mask* with respect to priority level l is the set of processors on which T_x is eligible to execute with priority at least l. For each lock-holding task, non-empty broadest masks are stored in order of decreasing priorities in the `squashed_mask_list`, which is similar in purpose to the top waiters list under normal priority inheritance. The `squashed_mask_list` of T_x is updated whenever a task joins or leaves the top-mask list of a lock that is currently held by T_x, and also when T_x acquires or releases a lock.

The `inheriting_list`. In Linux's current load-balancing implementation, the push/pull logic selects the target processor for a migrating task by comparing its effective priority with the current highest priority on *remote* processors. That is, the push/pull migration code considers the *local* priority of a task and compares with the priorities of remote tasks under the assumption that priorities do not change. Further, only the highest-priority task that is currently *not* scheduled is considered as a migration candidate to avoid scanning the entire run queue.

Unfortunately, neither is correct under migratory priority inheritance since a task's priority may vary on different processors. In fact, the current migration logic may consider a task with relatively low priority on one processor *never* eligible to migrate, even if its priority would be the highest on all the other processors. To overcome this issue, the migratory logic must *quickly* identify *all* tasks subject to migratory priority inheritance and check migrations opportunities for *each* for them, independently of their local priority.

To this end, tasks subject to migratory priority inheritance are kept track of by inserting them in a *dedicated* list on each run queue. This list (the `inheriting_list`) contains only migratory priority inheritance tasks that both are runnable, but not currently scheduled, and that have a non-empty `squashed_mask_list` (*i.e.*, preempted tasks that inherit some priority). Tasks in the `inheriting_list` are still subject to standard local scheduling, and are therefore still queued at their appropriate priority levels in the normal fixed-priority run queues. Furthermore, when a task is selected for scheduling, it should also be removed from the `inheriting_list`. (Similarly to the current push/pull migration logic, a scheduled task should never be considered for migration.)

The `inheriting_list` is employed by the push/pull migration logic to quickly enumerate those tasks that require special management. In addition, when tasks are moved to other run queues, it must be ensured that the `inheriting_list` is properly maintained and updated on each processor. Although each push/pull operation should consider each task in the `inheriting_list`, in a typical use case (*i.e.*, if locks are only infrequently contended), the `inheriting_list` should be short, or even empty, most of the time.

Local priority adjustment. If a lock-holding task inherits a higher-priority eligibility tuple that includes the processor where it is currently scheduled, its effective priority is immediately updated. Therefore, when the priority of a task T_x needs to be updated, the `__rt_mutex_adjust_prio()` function traverses the `squashed_mask_list` and updates the effective priority of T_x to the first priority that includes the current processor in the associated broadest mask. If no such priority level exists, then the scheduler is triggered to initiate a push migration (if applicable).

The priority adjustment logic must further update the normal fixed-priority run queues of the lock-holding task, as well as the `inheriting_list` if the task becomes scheduled due to the priority adjustment.

Push and pull migrations. Push and pull operations migrate a task that cannot be executed on the current processor (*e.g.*, because a higher priority task is already scheduled) to a processor where it has sufficient priority to be executed without delay.

Under migratory priority inheritance, every push/pull operation further attempts to migrate *each* task currently in the `inheriting_list`. For each such task T_x, a push/pull operation iterates over T_x's `squashed_mask_list` and, for each priority level l, tries to migrate T_x to one of the processors in the associated broadest mask. Since the `squashed_mask_list` is priority-ordered, a push/pull operation for a task T_x at priority level l can safely skip those processors where it was already not possible to migrate T_x at a higher priority.

For each lock-holding task subject to migratory priority inheritance, the above migration logic considers all eligibility tuples that contribute to the effective priority on some processor. If the lock-holding task has sufficient (inherited) priority to execute on some processor, the above migration logic will thus find the processor where the lock-holding task is eligible to execute immediately.

5.3 Implementation Challenges

While the changes discussed in the previous section are conceptually not very difficult, they do require considerable runtime management. In the following, we discuss potential bottlenecks and possible workarounds.

Update the `squashed_mask_list`. Unfortunately, updates to the `squashed_mask_list` are not balanced. Particularly, the compact representation used by the broadest-per-priority masks cannot be inverted, and does not allow to determine the individual contributions of each top mask waiter (as implied by "squashed").

Therefore, when a waiter leaves the top mask list of a lock held by a task T_x (*e.g.*, due to a timeout expiration or signal delivery), or when a lock is released by T_x, some or all of the broadest-per-priority masks may have to be recomputed starting from the top-mask lists of all the locks "reachable" from T_x in the undirected wait-for graph (*i.e.*, all the locks whose waiters are blocked by T_x). This expensive operation is the price to pay to enable efficient push/pull migrations. In fact, migration checks are far more frequent than (contended) lock releases or waiter cancellation and thus should only consider the tuples in the `squashed_mask_list` because said list is likely short and, in the worst case, bounded by the total number of priority levels.

Alternatives to the `squashed_mask_list` that do not require pre-computation of the broadest-per-priority masks may be equally expensive. For example, pre-computing the `squashed_mask_list` may be avoided by dynamically exploring the tree of all the top-mask lists for each of the reachable locks. Since this operation should be repeated at each push/pull migration attempt, for each task in the `inheriting_list`, the associated overheads can only be afforded if the lock-tree to explore is shallow and sparse.

An alternative for embedded systems. Another alternative implementation of migratory priority inheritance that does not rely on the `squashed_mask_list` is per-processor priority inheritance. This approach requires to split a task's priority field in an array of per-processor priorities. Normal priority inheritance is employed to independently update the highest priority values for the task on each (possibly inherited) eligible processor.

Although fast and efficient, memory usage is the major drawback of this approach. While it may be preferable for processor counts typically found in today's embedded systems (*i.e.*, roughly $\leq$ 32 processors), it clearly does not scale to large systems (such as the proverbial 4096-processor NUMA machine).

Updating the inheriting list. Under migratory priority inheritance, the `inheriting_list` allows to quickly identify tasks that require a special migration policy, and it is therefore fundamental to correctly realize the expected theoretical properties of migratory priority inheritance. Nonetheless, the `inheriting_list` is an additional per-processor list that, despite being frequently empty unless locks are permanently contended, needs to be appropriately managed. Furthermore, extra care is needed to enqueue and dequeue tasks in both standard priority run queues and in the `inheriting_list`. This causes additional overheads that can be avoided entirely by giving up on some of the analytical properties of migratory priority inheritance, as we discuss next.

5.4 A Simplified Implementation

In this section we briefly discuss a simplified version of migratory priority inheritance that trades improved efficiency for higher delays.

One of the main performance-critical operations of migratory priority inheritance is evaluating all the possible migration options in the `squashed_mask_list` when a task is selected of a push/pull migration.

The *simplified migratory priority inheritance* targets exactly this expensive operation. Under this variant, the `squashed_mask_list` is replaced by a single, unified *broadest mask* that holds the union of the affinity masks of *all* the top-mask waiters blocked by a lock-holding task, irrespective of their assigned priorities. Contrary to the `squashed_mask_list`, the broadest mask does not take priority levels into account, and simply represents the set of processors on which the lock-holding task is entitled to run, at either its base or an inherited priority.

Under simplified migratory priority inheritance, a lock-holding task is allowed to execute on *every* processors included in its broadest mask, at its *highest* (inherited) priority. Contrary to full migratory priority inheritance, simplified migratory inheritance preserves both current Linux's top waiter semantics and top-waiters list. In fact, with the above relaxed execution rule, simplified migratory priority inheritance behaves exactly like normal priority inheritance with regard to priorities. Since the priority of a task does no longer depend on the processor where it is scheduled, it should not be re-evaluated as part of each push/pull migration, but only—as under classic priority inheritance—when the top waiters list is updated.

Further, under simplified priority inheritance, the effective priority of a task is a *global* property that is valid on all processors where the task is eligible to execute, and therefore, push/pull migrations can safely compare the "local" priority of a task with remote priorities. The lack of a global effective priority was the motivating reason for the `inheriting_list` under full migratory priority inheritance. This additional list is therefore not required under simplified priority inheritance.

The only changes to the current Linux's priority inheritance support are related to the insertion of waiters in the *top mask* lists for each lock, and to the update of the broadest affinity mask of the lock-holding task. When new waiters joins the top mask list of a lock, updates to the broadest masks are considerably faster than those to the `squashed_mask_list`. Nonetheless, the same challenges noted for updates to the `squashed_mask_list` also apply to the broadest mask when a waiter leaves the top mask list of some lock, or when a lock is released. In these cases, the broadest mask should be recomputed from the top mask lists of all the locks "reachable" by the current lock-holding tasks.

Compared to full migratory priority inheritance, the above enhancements on the implementation side entail higher delays for a larger number of tasks. In fact, under full migratory priority inheritance, only critical sections that are accessed by "potentially local" equal-or-higher-priority tasks can cause delays. (A task is "potentially local" to another task if their affinity masks are not fully disjoint.) Instead, under the simplified variant, tasks can also be delayed by critical sections accessed by all equal-or-higher-priority tasks in the system.[5]

In fact, since a lock-holding task can run at the priority of its topmost waiter task on any processors in the broadest mask, it can possibly delay all those tasks with lower priority than the topmost waiter, and whose affinity masks are "connected" to that of the topmost waiter by the affinity mask of any other waiter of the lock. (The affinity mask does not directly overlap with the topmost waiter, but the two tasks are "connected," since both their affinity masks overlap with the one of a common third waiter of the lock.)

Despite these higher delays, it is important to note that, like full migratory priority inheritance, simplified migratory inheritance bounds priority inversion in all cases and is thus analytically sound and predictable. Furthermore, since it considerably easier to implement and since it leaves the push/pull logic unchanged, it could be a worthwhile tradeoff between analytical properties and practical efficiency. However, we caution that whether the simplified or full variant of migratory priority inheritance is used is not merely an "implementation detail," as the potential for increased blocking under the simplified variant must be taken into account during schedulability analysis.

6 Related Work

In this section, we relate migratory priority inheritance to similar concepts and give a brief overview over major real-time locking protocols for uni- and multiprocessors.

Uniprocessors. Besides classic priority inheritance [39, 41], there are two major real-time locking protocol alternatives for uniprocessors. Both the *priority-ceiling protocol* (PCP) [39, 41] and the *stack resource policy* (SRP) [7] ensure that a real-time job is blocked by at most one outermost critical section across *all* locks (and not *per* lock, as under priority inheritance, assuming the job does not self-suspend for locking-unrelated reasons), which is optimal. To accomplish this, these protocols require additional information: for each lock, a *priority ceiling* must be specified, which is the highest priority of any task that accesses the lock. Determining a lock's priority ceiling can be difficult in complex systems such as Linux, which makes the SRP and PCP somewhat less convenient to use in practice. Further, the PCP and the SRP require the kernel to keep track of which locks are currently held even if locks are not contended, which creates additional overheads.

The POSIX standard mandates the availability of what is essentially the SRP (under the name PRIO_PROTECT),

[5] Note that this is still less disruptive than priority boosting, where any critical section can delay any real-time task regardless of priority.

which is hence also supported in Linux. However, Linux's PRIO_PROTECT emulation uses system calls to adjust a task's effective scheduling priority at the beginning and end of each (outermost) critical section and thus incurs considerably higher average-case overheads than Linux's carefully tuned priority inheritance implementation (PRIO_INHERIT).

Multiprocessors. To enable predictable locking in multiprocessor real-time systems, various real-time spinlock and semaphore protocols have been proposed, with support for mutual exclusion [4, 9, 20, 21, 26, 30, 35, 37–40, 42–44], reader-writer exclusion [12, 14–16], and k-exclusion [15, 16, 22, 46]. Recently, protocols for predictable lock nesting [45] have been proposed as well.

Spinlock protocols [4, 9, 12, 14, 20, 26, 42–44] have several advantages: they are easy to analyze and implement, they cause relatively little overhead since waiting tasks do not suspend, and as a result can yield superior schedulability compared to semaphore protocols [10, 11, 17]. However, most spinlock protocols require spinning jobs to be non-preemptive, which implies that critical sections must be short. In the context of Linux, this renders non-preemptive spinlocks unsuitable for all but a few select locks in the core kernel (*e.g.*, run queue locks). While preemptable spinlocks [4, 42–44] have been studied as well, such locks are typically more complicated than their non-preemptive counterparts, which detracts from the main advantage of spinlocks, namely simplicity.

Semaphore protocols for real-time multiprocessor systems can be categorized by the scheduling approach for which they have been developed. Several real-time locking protocols based on priority inheritance have been devised specifically for global scheduling [9, 13, 16, 21, 22, 35, 46]. Since these protocols are based on classic priority inheritance, they are subject to the limitations described in this paper (recall §3.2) and thus do not close the "predictability gap" that we seek to address in this paper.

More relevant to this paper are real-time locking protocols for partitioned and clustered schedulers [9, 15, 16, 30, 37–40], which ensure bounded priority inversions under non-global scheduling. However, each of the just-cited locking protocols employs some variant of priority boosting and is thus liable to increase worst-case latencies (recall §3.4). As we have argued before, any protocol based on priority boosting is fundamentally inappropriate for Linux due to the uncertainty surrounding maximum critical section lengths in lower-priority tasks.

Helping in Fiasco. A mechanism closely resembling migratory priority inheritance appeared in TU Dresden's Fiasco microkernel already in 2001 under the name *local helping* [27, 28]. The mechanism derives its name from the fact that the lock-holding task is migrated to the blocked task's processor, where it is "locally helped" to finish its critical section. We prefer the name "migratory priority inheritance" over "helping" because "helping" could be easily misunderstood to refer to wait-free synchronization algorithms, of which some also employ "helping" to complete pending updates (*e.g.*, see [5]). We note, however, that local helping was introduced in Fiasco for virtually the same motivation as pointed out here, namely to reduce blocking under P-FP scheduling.

A major difference between Fiasco's local helping and migratory priority inheritance is that Fiasco employs polling (*i.e.*, spinning) while the lock-holder has not (yet) been preempted. In Fiasco, this is reasonable since lock-holding tasks never suspend [28]. In Linux, however, tasks holding mutexes do potentially self-suspend; busy-waiting thus risks wasting considerable processor capacity.[6] Another difference is that migratory priority inheritance, as defined in §4.1, works seamlessly with arbitrary processor affinity masks, whereas local helping as described by Hohmuth and Peter [28] is specific to P-FP scheduling.

Bandwidth inheritance. Most relevant to this paper, and to migratory priority inheritance in particular, is the concept of *bandwidth inheritance* [24, 31]. Bandwidth inheritance was originally devised by Lamastra *et al.* [31] to transfer the concept of priority inheritance to *reservation-based scheduling*, where each task's rate of execution is limited by a periodically replenished execution budget. On a uniprocessor, bandwidth inheritance primarily solves the problem of excessive blocking that can arise when lock holders exhaust their budget. To this end, a lock holder that has exhausted its budget may consume the budget (or *bandwidth*) of tasks that it blocks. Faggioli *et al.* extended bandwidth inheritance to multiprocessors [24]. Under their *multiprocessor bandwidth inheritance protocol* (MBWI), blocked tasks preemptively spin (from an analysis point of view) while the lock holder is scheduled, but make their processor available to the lock holder if it is preempted or runs out of budget. That is, as in the uniprocessor case, lock holders may use the budget of any waiting task, but must first migrate to a waiting task's processor prior to consuming its budget.

Migratory priority inheritance, as proposed in this paper, is inspired by the MBWI, but differs in a few important points. Notably, tasks never busy-wait under migratory priority inheritance, which matches Linux's current locking semantics. Further, the MBWI fundamentally assumes the presence of reservation-based scheduling, which is currently not available in Linux (though a high-quality option is available with the SCHED_DEADLINE patch [23, 32]). In contrast, migratory priority inheritance

[6] Note that Fiasco is a microkernel in which synchronization is mostly achieved using inter-processor communication (IPC). Locking occurs only in the microkernel itself (*e.g.*, when updating process control blocks) and thus involves only very short critical sections. In contrast, locks are pervasively used in both user and kernel space in Linux, and long critical sections cannot be ruled out.

can be applied to Linux as it works today, and could still be combined with reservation-based scheduling should it be included in the future. Finally, while the MBWI transparently supports partitioned, clustered, and global scheduling, it does not address the "predictability gap" in the fixed-priority-based POSIX environment since it assumes EDF scheduling. In a sense, migratory priority inheritance can be understood to be a simplified variant of Faggioli *et al.*'s MBWI [24] that has been reduced to the core mechanism required to bound priority inversions.

7 Conclusion

We have examined Linux's support for predictable locking primitives and found that Linux is currently well-equipped for hosting real-time workloads on uniprocessors only, but not on multiprocessors. We have discussed how this "predictability gap" stems from the fact that classic priority inheritance is ineffective under non-global scheduling. As a solution, we have proposed to employ migratory priority inheritance instead, which we believe to be a good fit for Linux for four key reasons:

1. it is a straightforward generalization of priority inheritance that can be incorporated without substantial changes into the existing kernel infrastructure for futexes, priority inheritance, and task migrations;

2. it maintains the desirable property that the scheduling latency of high-priority tasks is not affected by critical sections of lower-priority tasks;

3. it neither breaks POSIX compliance nor conflicts, in our opinion, with application developer expectations; and

4. it reliably bounds the worst-case duration of priority inversion to one critical section length (each time that a lock is accessed).

In short, migratory priority inheritance is a simple mechanism that closes the "predictability gap" without fundamentally altering Linux's locking and scheduling behavior.

Considering implementation concerns, we have discussed the design of a prototype implementation and believe that migratory priority inheritance can be realized with reasonable effort and acceptable overheads. Nonetheless, for the case that overheads are deemed too large, we have also discussed a simplified variant of migratory priority inheritance, which, as a trade-off, offers reduced implementation-related overheads at the cost of slightly increased worst-case blocking. While a production-quality implementation of either full or simplified migratory priority inheritance would certainly require additional work and testing, it is our hope that the proposed prototype may serve as a basis for discussion.

We are currently working on schedulability analysis for locking protocols based on migratory priority inheritance under a number of different real-time scheduling policies (including clustered and partitioned scheduling using either fixed or EDF priorities) and analysis assumptions (*e.g.*, "suspension-oblivious" analysis, see [13]). In future applied work, we plan to evaluate migratory priority inheritance in LITMUS^{RT} [1] in terms of schedulability under consideration of overheads. Further, for migratory priority inheritance to be considered in practice, it will be necessary to carefully study its effects on throughput and observed average- and worst-case response times.

References

[1] The LITMUS^{RT} project web site. `http://www.litmus-rt.org`.

[2] *IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP) - POSIX Realtime and Embedded Application Support*. Number Std 1003.13-2003. IEEE Computer Society, 2003.

[3] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Base Specifications*. Number Std 1003.1-2008. IEEE Computer Society, 2008.

[4] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355, 1998.

[5] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing*, pages 229–238. ACM, 1997.

[6] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

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

[8] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*. Chapman Hall/CRC, 2007.

[9] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for

multiprocessors. In *Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.

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

[11] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In *Proceedings of the 12th International Conference On Principles Of Distributed Systems*, LNCS 5401, pages 105–124. Springer-Verlag, 2008.

[12] B. Brandenburg and J. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *Proceedings of the 21th Euromicro Conference on Real-Time Systems*, pages 184–193, 2009.

[13] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st Real-Time Systems Symposium*, pages 49–60, 2010.

[14] B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87, 2010.

[15] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the 9th ACM International Conference on Embedded Software*, 2011.

[16] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, to appear, 2012.

[17] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, 2008.

[18] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256, 2007.

[19] J. Corbet. Priority inheritance in the kernel. *Linux Weekly News*, `http://lwn.net/Articles/178253/`, 2006.

[20] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 75–84, 2006.

[21] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 377–386, 2009.

[22] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems. In *Proceedings of the 19th International Conference on Real-Time and Network Systems*, pages 15–24, 2011.

[23] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An EDF scheduling class for the linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop*, 2009.

[24] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 90–99, 2010.

[25] H. Franke, R. Russel, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the 2002 Ottawa Linux Symposium*, pages 479–495, 2002.

[26] 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 *Proceedings of the 9th IEEE Real-Time And Embedded Technology Application Symposium*, pages 189–198, 2003.

[27] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, 2001.

[28] M. Hohmuth and M. Peter. Helping in a multiprocessor environment. In *Proceeding of the Second Workshop on Common Microkernel System Platforms*, 2001.

[29] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[30] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 469–478, 2009.

[31] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 151–160, 2001.

[32] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2011.

[33] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[34] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, 1973.

[35] G. Macariu and V. Cretu. Limited blocking resource sharing for global multiprocessor scheduling. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 262–271, 2011.

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

[37] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 251–261, 2011.

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

[39] R. Rajkumar. *Synchronization In Real-Time Systems—A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[40] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

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

[42] H. Takada and K. Sakamura. Predictable spin lock algorithms with preemption. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 2–6, 1994.

[43] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 134–143, 1997.

[44] C.-D. Wang, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 70–76, 1996.

[45] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 223–232, 2012.

[46] B. Ward, G. Elliott, and J. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *Proceedings of the 18th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 280–289, 2012.