Bringing Theory Into Practice: A Userspace Library for Multicore Real-Time Scheduling

Malcolm S. Mollison and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

As multicore computing hardware has become more ubiquitous, real-time scheduling theory aimed at multicore systems has become increasingly sophisticated and diverse. Real-time operating systems (RTOSs) are ill-suited for this kind of rapid change, and the slow-moving RTOS ecosystem is falling further and further behind advances in real-time scheduling theory. Thus, supporting new functionality in a layer of middleware software running in userspace (i.e., outside the RTOS kernel) has been proposed. In this paper, we describe the first userspace scheduler that supports preemptive, dynamic-priority, migrating real-time tasks on multicore hardware, and report empirical latency and overhead measurements. On an eight-core Intel Xeon platform, these measurements are in the range of ones to tens of microseconds under most tested configurations. We believe that this approach may prove superior to a kernel-based approach for supporting a subset of future real-world real-time applications.

1. Introduction

In recent years, real-time systems researchers have developed an increasingly complex and diverse ecosystem of scheduling algorithms and locking protocols. Particularly large strides have been made with regard to resource allocation techniques targeting multicore systems. Unfortunately, industry practitioners wishing to begin making use of these advancements today are generally unable to do so, because software infrastructure to support these advancements remains largely unavailable.

The majority of implementation-oriented research in this area has focused on modifying RTOS kernels to support new resource allocation techniques. For example, recent work has shown that the *clustered earliest-deadline-first algorithm* (C-EDF) performs well on large multicore machines [26], and an asymptotically optimal locking protocol that can be used with this algorithm has been given [25].

Kernel-focused work has been invaluable in demonstrating the capabilities and limitations of new multicore resource allocation techniques on actual hardware. However, our prior work with colleagues in industry suggests that adopting this approach in deployed systems is unappealing, and that a userspace (*i.e.*, middleware) approach may be much more readily useful, if such an approach were to prove feasible. Below, we list some of the reasons for this observation.

1) *Customization.* Industrial practitioners would benefit from the ability to select and deploy resource allocation techniques commensurate with their particular applications, rather than being "shoehorned" into the relatively "one-size-fits-all" traditional commercial/open-source RTOS software model. RTOSs have been very slow to change, and cannot easily adapt to the growing diversity of resource allocation techniques.
2) *Robustness.* A middleware approach could potentially allow different resource allocation techniques to be employed by different groups of co-hosted applications, assuming these groups do not share cores with one another. This would permit critical and/or legacy applications to run independently on the underlying, unmodified RTOS. In contrast, a modified RTOS kernel would expose all co-hosted applications to (potentially unacceptable) risk from defects in resource allocation software.
3) *Maintainability.* Well-designed middleware could potentially run without modification from one underlying OS version to the next. In contrast, a kernel-based approach entails time-consuming and potentially unsafe modifications every time existing software infrastructure is deployed on a newer OS version.
4) *Portability.* Similarly, well-conceived middleware software could be easily ported between entirely different OSs, whereas a kernel-based approach would typically require starting nearly from scratch for each new OS. The value of software infrastructure greatly decreases when much effort must be spent to allow it to run on a new OS, so portability tends to be highly valued among industry practitioners.
5) *Historical precedent.* While the core functionality of most RTOSs has remained mostly unchanged in recent decades, various kinds of middleware have seen widespread adoption in industry. For example, [1] lists over fifty real-world applications using the TAO [7] distributed communication middleware.

The potential usefulness of a middleware approach raises the question, thus far largely unexplored, of whether middleware can support recent multicore real-time resource allocation techniques. Specifically, most such techniques require

Work supported by the NC Space Grant College and Fellowship Program; NC Space Grant Consortium; NSF grants CNS 1016954 and CNS 1115284; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

fully-preemptive tasks, the priorities of which can be changed dynamically at runtime.

Contributions. In this paper, we present a userspace library that can support fully-preemptive multicore scheduling of dynamic-priority real-time tasks, thereby answering in the affirmative the question of whether middleware can support state-of-the-art real-time scheduling techniques. In our library, real-time tasks are instantiated as user-level threads sharing a single address space. We also present empirical measurements of overheads and latencies for the library that generally fall into the range of ones to tens of microseconds (as seen in Tables 1 and 2). Under more taxing configurations, wherein tasks are permitted to migrate among four or more cores, worst-case latencies range into the low hundreds of microseconds.

Overheads and latencies for kernel-based implementations of similar real-time techniques fall into similar ranges [26]. However, making a direct comparison between these two approaches is not straightforward, as discussed in Section 4.

We chose to focus on a user-level threading solution, as opposed to a more heavyweight solution that would support memory protection between tasks, for two reasons. First, we believe that a user-level threading solution is likely to achieve lower overheads and latencies than any alternative middleware-based approach (such as manipulating the scheduling parameters of kernel-level threads). Second, sharing a single address space is standard practice for concurrent threads within a single application, which is the specific scenario that prompted this work (as described immediately below). However, our approach does permit memory protection between real-time tasks belonging to different applications that are assigned to disjoint sets of cores.

Our interest in this topic arose from interactions with industry colleagues developing next-generation unmanned aerial vehicles (UAVs), which will be far more capable than current UAVs, particularly with regards to autonomy. The design of our library would allow novel applications to use innovative scheduling techniques across certain fixed cores of a multicore machine. Given sufficiently capable scheduling support, these applications could incorporate real-world domain knowledge into scheduling decisions. For example, real-world deadlines could be used to dynamically set the deadlines of real-time tasks. Meanwhile, more conventional applications with static priorities could be safely scheduled by the underlying RTOS on other cores.

In accordance with the desired properties listed above, the library relies upon only a small number of operating system features and is believed to be easily maintainable (across different versions of a given OS) and portable (to different OSs). In our evaluation we employed the popular `PREEMPT_RT` Linux kernel variant as the underlying RTOS.

The source code of the library is available online [11].

Relationship to prior work. To the best of our knowledge, there exists no other userspace resource allocation software that supports as general a class of resource allocation techniques as that presented herein—specifically, those wherein tasks are fully preemptive, have dynamic priorities, and can migrate between cores. However, a number of more restricted classes have been supported previously at the user level, and the same class has been supported at the kernel level. In earlier work [38] we raised the possibility of creating a library such as that presented here.

The rest of this paper is organized as follows. In Section 2, we provide background information. In Section 3, we describe the implementation of our library. In Section 4, we present an empirical evaluation of the library (particularly, measured overheads). Finally, in Section 5, we discuss future work and conclude.

2. Background

In this section, we first describe the sporadic task model, which is the resource model targeted by our library. This model is identical to or more general than that assumed by most research on multicore real-time systems. Then, we describe the resource allocation techniques (scheduling algorithms and synchronization protocols) that the library targets. Finally, we describe related real-time resource allocation software.

2.1. Sporadic Task Model

The basic unit of computational work is a series of sequential instructions known as a *task*. In a real-time task system, a *sporadic task* T has an associated *worst-case execution time* (WCET), $T.e$, and *minimum separation time*, $T.p$. Each successive *job* of T is released at least $T.p$ time units after its predecessor. The *utilization*, or long-run processor share required by a sporadic task, is given by $T.u = T.e/T.p$. Associated with each sporadic task is a *relative deadline*, $T.d$. In an *arbitrary-deadline* task system, task deadlines may be be greater than, equal to, or less than $T.p$. A notable special case of a sporadic task is a *periodic task*, wherein each successive job is released precisely $T.p$ time units after its predecessor.

A task system of n tasks is *schedulable* if, given a scheduling algorithm and m processors, the algorithm can schedule tasks in such a way that all of their timing constraints are met. More specifically, for *hard real-time* task systems, jobs must never miss their deadlines, while for *soft real-time* task systems, some deadline misses are tolerable. A common interpretation of soft real-time correctness is that the tardiness of jobs of soft real-time tasks must be bounded by a pre-computed constant that is reasonably small.

2.2. Scheduling Algorithms

Approaches to scheduling real-time tasks on multicore systems can be categorized according to two fundamental (but related) dimensions: first, the choice of how tasks are mapped onto processing cores; and second, the choice of how tasks are prioritized. In each case, there are two common choices. Tasks are typically mapped onto cores either by *partitioning*, in which each task is assigned to a core at system design time and never migrates to another core; or by using a *migrating* approach, in which tasks are assigned to cores at runtime (and can be dynamically reassigned). Tasks are typically prioritized using either *static priorities*, in which case priorities are chosen at design time and never change; or *dynamic priorities*, in which case

tasks' priorities relative to one another change at runtime according to some criteria specified by the scheduling algorithm. Related to task prioritization is whether tasks are *non-preemptible*, in which case a higher-priority task may have to wait for a previously-scheduled lower-priority task to complete before being scheduled; or *fully-preemptible*, in which case the highest-priority task is always scheduled. Our userspace scheduling library aims to support migrating, dynamic-priority, fully-preemptive scheduling algorithms.

An example of such an algorithm is the clustered earliest-deadline-first (C-EDF) algorithm, which was mentioned in Section 1. Under C-EDF, before system runtime, each core is assigned to one of c clusters, where $1 \leq c \leq m$; and each of the n tasks is assigned to one of the clusters.[1] For simplicity, we assume a uniform cluster size, d, given by m/c. At system runtime, within each cluster, the d eligible tasks with highest priority are scheduled on the d available cores. Eligible tasks are those that have a released job and are not waiting for a shared resource to become available.

As of the time of writing, C-EDF is the only scheduling algorithm implemented by our userspace scheduling library. This implementation is the basis of the empirical measurements provided in Section 4. We chose to focus on C-EDF because it exhibits many of the key features of other migrating, dynamic-priority, fully-preemptible scheduling algorithms. C-EDF provides the basic foundation for many such algorithms.

A typical C-EDF implementation maintains three key data structures for each cluster.

1) The *ready queue* contains tasks that are eligible, and is ordered in earliest-deadline-first order.
2) The *release queue* contains tasks that are ineligible but will release a job in the future, and is ordered in earliest-release-time-first order.
3) The *priority mapping* tracks the priority of the task running on each processor, and is referred to in order to preempt the lowest-priority task when a higher-priority task is released.

2.3. Synchronization Protocols

A real-time *synchronization protocol* is used to arbitrate among tasks that share resources that cannot be simultaneously accessed by any number of tasks, such as a critical section of code or a shared hardware device. These protocols typically attempt to reduce *priority inversions*, in which lower-priority tasks are allowed to execute in favor of higher-priority tasks due to resource-sharing dependencies. The possibility of priority inversions in a system must be accounted for in schedulability analysis. A number of multiprocessor locking protocols have been developed (for example, [41,42]). To demonstrate that our approach is compatible with synchronization protocols, our library includes support for the global version of the *flexible multiprocessor locking protocol* (FMLP) [23]. Our FMLP implementation supports both "short" (spin-based) and "long" (suspension-based) synchronization.

1. Under C-EDF, tasks are partitioned (and do not migrate) if and only if $c = m$.

2.4. Related Software

There is a large body of pre-existing resource allocation software. We present a survey of this body of work below. Due to space constraints, we omit discussions of microkernel-specific approaches, real-time programming languages, and non-real-time parallel threading libraries.

RTOSs. Typical commercial and open-source RTOSs support only static-priority scheduling. This stands in contrast to our library, which aims to support dynamic-priority scheduling policies. One notable exception is the ERIKA Enterprise RTOS [3], which supports partitioned EDF scheduling. A classification and summary of existing RTOSs can be found in [26].

Kernel patches. LITMUS$^{\mathrm{RT}}$ [6,28] patches the Linux kernel to provide support for the same class of schedulers and locking protocols that is targeted by our library. Thus, our library can be viewed as a userspace analogue to LITMUS$^{\mathrm{RT}}$, though LITMUS$^{\mathrm{RT}}$ is at a far more mature stage of development. Practitioners seeking to use LITMUS$^{\mathrm{RT}}$ to support real-world workloads would face the pitfalls delimited in Section 1. For these and other, related reasons, LITMUS$^{\mathrm{RT}}$ is intended to be a research tool, and is not intended for practical deployment.

Another Linux kernel patch, `SCHED_DEADLINE` [10,33, 36], targets C-EDF specifically. For more than two and a half years, its developers have been working with Linux kernel contributors to have it adopted for eventual mainlining in the Linux kernel. The longevity of the effort to mainline this project—which supports only a single scheduling algorithm—helps to demonstrate the points made in Section 1.

Kernel modules. AQuoSA [39] provides a real-time resource reservation mechanism for Linux. While AQuoSA requires a kernel patch, it decreases coupling with the kernel by moving some functionality into a *kernel module*, which is compiled independently of the kernel, but still executes in kernelspace (*i.e.*, inside the kernel).

A promising line of recent work has investigated kernel-module-based scheduling systems that allow customized schedulers provided from userspace to be loaded as plugin components. The most recent representative is ExSched [21].

Compared to such an approach, the approach we present in this paper has the downside of being limited to scheduling within a single application, instead of among multiple applications instantiated as separate processes. However, in our domain of interest, it is important that applications with novel, customized, or domain-specific schedulers be allowed to fail independently of one another in the presence of bugs in scheduler code. This property allows such applications to run alongside safety- or mission-critical applications on the same machine. Moreover, our approach more completely decouples resource allocation code from the kernel, and thereby more completely addresses the pitfalls mentioned in Section 1.

Scheduler activations. *Scheduler activations* were proposed in 1992 [20] as an alternative to conventional kernel-level threads that effectively allow user applications to define application-specific scheduling policies. Were scheduler ac-

tivations to be present in modern RTOSs, the design of our library (as described in Section 3) could be significantly simplified. Scheduler activations have been offered most recently in certain versions of the Solaris operating system and in Windows 7; unfortunately, we are not aware of any modern RTOSs that offer scheduler activations.

DRE middleware. There exists a wide body of work on *distributed, real-time, embedded* (DRE) middleware, which offers functionality beyond that provided by the underlying RTOS [35]. This work differs from the library presented here primarily in that real-time tasks are either non-preemptive, or are preemptive only across the static priority levels offered by the underlying RTOS. DRE middleware has seen widespread adoption in industry [1]. This supports our hypothesis that a userspace approach holds promise for bringing the resource allocation techniques supported by our library to industry practitioners.

Preemptive userspace libraries. There also exists prior work on userspace-based resource allocation techniques that do offer preemptivity, but which, for other reasons, fall outside the class of techniques supported by our library.

The most relevant example that we are aware of was presented in a 2004 investigation of EDF on power-constrained systems that relied on a preemptive userspace scheduler [18, 19]. Another is `wuthreads` [14], which was created as a pedagogical demonstration of userspace scheduling.

These examples are less sophisticated than the library presented here, and are not aimed at investigating or providing real-world usability. For example, neither project supports multicore scheduling, includes empirical evaluations of overheads and latencies, addresses portability, supports system calls in a useful manner, or offers support for synchronization protocols.

Virtualization. RTAI [30], RT-Linux [43], and Xenomai [15] permit a real-time kernel and the Linux kernel to be co-hosted by virtualizing interrupt delivery. VSched [37] is a userspace Linux system that provides resource reservation services for virtual machines. LinSched [27] is an application that allows Linux kernel scheduler code to be simulated, tested, and debugged in userspace, albeit in a single-threaded manner.

3. Implementation

In this section, we first describe the overall architecture of our library. In subsequent subsections, we describe how the library addresses three critical design challenges: providing preemptivity; supporting critical sections within the scheduler; and facilitating system calls. Finally, we discuss miscellaneous remaining issues.

3.1. Overall Architecture

In order to schedule real-time tasks completely independently of the kernel, the library makes use of a classic design pattern in computer science: the introduction of an additional level of indirection. This is composed of two basic building blocks, kernel-level threads and user-level threads, as described below. A schematic of the architecture is shown in Figure 1.

Kernel-level threads. A given real-time application is allocated a set of processors available on the system. A kernel-level thread (*i.e.*, an entity directly schedulable by the native kernel scheduler) is spawned for each processor allocated to the application. Each kernel-level thread is pinned to a specific, unique processor. The purpose of the kernel-level threads is to act as "virtual CPUs" that can execute real-time tasks. We denote these threads as *worker threads*. Each worker thread is assigned the highest available real-time priority in the system.[2] Thus, on a correctly implemented kernel, worker threads are only suspended due to system calls they themselves issue (which are carefully controlled). This allows our library to guarantee that the worker threads are always available to process work, and thereby ensure real-time timing constraints.

User-level threads. The real-time tasks that make up the application are instantiated as user-level threads (*i.e.*, entities not visible to the native kernel scheduler, but instead controlled directly by kernel-level threads—in this case, the worker threads). User-level threading is a technique that was commonly used by threading libraries before kernel-level threads were widely available [32], and that is traditionally made available by POSIX[3]-compliant C libraries. In user-level threading, the current execution context of a CPU can be stored to memory, and a previously-stored context can be loaded onto the CPU.

In our library, when a user-level thread begins executing for the first time, a library-defined harness function is called. This function then calls an application-provided function that serves as an entry point for the real-time task corresponding to that user-level thread.

Scheduling. Our library defines a function, `schedule()`, which is responsible for performing any context switch from one user-level thread to another. In our C-EDF scheduler, it switches execution to the earliest-deadline task in the ready queue, if that task has an earlier deadline than the task already running in a particular worker thread.

3.2. Preemptivity

The architecture described thus far is sufficient to enable non-preemptive C-EDF scheduling (with a minor modification to `schedule()` to check the release queue for newly-eligible tasks). In this subsection, we describe one of the key novelties of our library: enabling preemptive scheduling.

Release timer. In C-EDF, the need for preemptivity arises due to job releases. Specifically, a task may become eligible due to a new job release at an arbitrary point in time; if it has an earlier deadline than any currently-running task, it must be scheduled immediately.

A POSIX timer is used to deal with this situation. At system initialization, the timer is set to fire at the time of the earliest job release. Each time the timer fires, it is re-armed

2. With the possible exception of interrupt bottom halves and proxy threads; both are discussed later.

3. A family of IEEE standards for compatibility between operating systems.

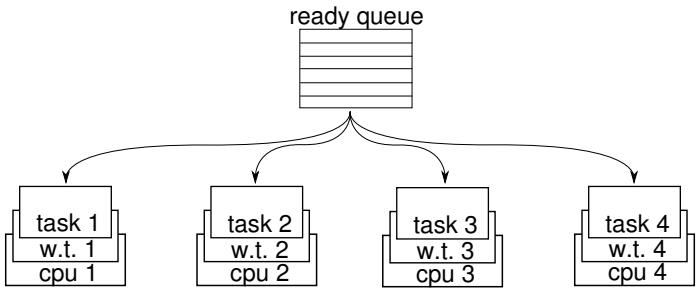

Figure 1: Schematic of overall architecture. This application is scheduled across one cluster of four processors. "w.t." stands for "worker thread." Note how worker threads perform the role of "virtual CPUs" for real-time tasks. Auxiliary data structures (most notably, the release queue) are not pictured.

to fire again at the time of the subsequent job release.

Handling releases. Each time the release timer fires, a POSIX signal of type `RELEASE_SIGNAL` (a library-defined constant) is emitted. The emission of this signal causes the library-defined function `release_handler()`, a *signal handler*, to run asynchronously in one of the worker threads (*i.e.*, it will interrupt the currently-executing real-time task). In addition to re-arming the release timer, this function has three responsibilities. First, it moves any tasks that are newly eligible from the release queue to the ready queue. Second, if any of these newly-eligible tasks necessitate re-scheduling on remote worker threads, it emits POSIX signals of type `PREEMPTION_SIGNAL` directly to the appropriate threads using the `pthread_kill()`[4] function, as is discussed in greater detail below. Finally, if local re-scheduling is needed, it calls `schedule()`.

Handling preemptions. Like the `release_handler()` function, the `preemption_handler()` function is a signal handler that executes asynchronously in a worker thread. It is called only in response to a signal of type `PREEMPTION_SIGNAL` emitted from an invocation of `release_handler()`.

3.3. Scheduler Critical Sections

Observe that both the `schedule()` and `release_handler()` functions are critical sections, because they update per-cluster data structures (as described in Section 2.2). To guard these data structures against simultaneous access by multiple worker threads, they are protected by a user-level spin lock. (The implementation of spin locks in our library is discussed later.)

Deadlock prevention protocol. Absent further precaution, the system as described thus far would be prone to deadlock. That is because a worker thread holding a per-cluster lock could be asynchronously interrupted by either of the two signal handlers. The thread would then attempt to re-obtain the per-cluster lock, without having first released it.

To prevent asynchronous interruptions in these two critical sections, two techniques are used. First, when one of these critical sections is entered via a signal handler, signals are automatically blocked upon signal handler invocation, and automatically unblocked upon signal handler return. POSIX allows this behavior to be configured for each signal handler. Second, when one of the critical sections is entered synchronously (*e.g.*, when a job completes, `schedule()` is called), the library explicitly blocks signals using an appropriate POSIX C library call. When the critical section is exited, signals are explicitly unblocked. Note that signal blocking and unblocking is specific to each worker thread.

Safe signal handler postponement. When a signal handler is invoked, causing a new task to be scheduled, the signal handler will not return until the task it interrupted is eventually re-scheduled. Before that task is re-scheduled, many other tasks may run in the meantime. When it finally is re-scheduled, the signal handler will return; however, it could return in a different worker thread than the one from which it was invoked. Counterintuitively, this behavior cannot cause signals to be blocked for an arbitrary amount of time while a signal handler is outstanding, which would cause temporal failure. That is because the deadlock prevention protocol described above guarantees the following invariant: whenever a context blocks signals, the same context or a different context will immediately unblock them again after the critical section has completed.

3.4. System Calls

Under POSIX, when a signal is received by a kernel-level thread that is performing a system call, the system call may immediately return with an error. In other cases, the system call proceeds after the signal handler returns, but may cause faulty behavior if any *async-signal-unsafe* system calls are invoked in the meantime [16]. Because of the continual use of signals by the library (and postponed return of signal handlers), our library treats system calls specially, in one of two possible ways (as described below).

Proxy threads. For system calls that can cause blocking in the kernel for a long or unbounded duration (*e.g.*, waiting for data to arrive over a socket), real-time tasks should request entrance to a *proxy thread*. Proxy threads are special kernel-level threads that are spawned (in any application-defined quantity) during application initialization. Proxy threads are prioritized immediately above worker threads. Normally, proxy threads are suspended, waiting for a special `PROXY_ENTER` signal.

4. `pthreads` is the POSIX API for thread control.

To request use of a proxy thread, a real-time task invokes a library function that causes the task to be de-scheduled, and then emits a `PROXY_ENTER` signal. Upon receipt of the signal, the proxy thread is scheduled by the kernel scheduler and initiates a context switch to the real-time task requesting its services. The real-time task should immediately make the system call. If the system call blocks, the proxy thread is suspended by the kernel, and if a worker thread was assigned to the same processor, it is resumed. When the system call returns, the proxy thread is again scheduled, and the real-time task should immediately request an exit from the proxy thread. The real-time task's context is saved, and a special `PROXY_EXIT` signal is sent to a worker thread to request that the real-time task be returned to the ready queue.

Communication between proxy threads and worker threads is problematic: shared data structures implemented using locks could cause deadlock if a proxy thread were to resume on the same processor as a worker thread holding the lock. To resolve this issue, some form of lock-free communication must be utilized. In our implementation, the `task_id` is communicated between proxy threads and worker threads by "attaching" it to the signals that are exchanged, using a special POSIX feature offered by the `sigqueue` function.

Proxy threads are not problematic for real-time schedulability analysis, because potential interruptions of worker threads by proxy threads (which are short) can be treated as non-preemptive critical sections.

Short system calls. For system calls that do not block or are likely to only block for a short period of time, it is sufficient for the real-time task to simply block signals before and after the system call. This may cause the worker thread to become suspended, and temporarily unable to serve other tasks. Analytically, such a system call can be treated as a non-preemptive critical section.

3.5. Other Issues

Other implementation details that may be of interest to some readers are discussed in this subsection.

Accessing time. To enable extremely high-resolution overhead measurements and scheduling decisions to be made, our library relies on an x86-architecture-specific feature known as the *timestamp counter* (TSC). The TSC is a per-processor register that records the number of CPU cycles that have elapsed since the processor was initialized at boot time. Using the TSC makes inaccuracies in time measurement a non-issue for our library.

We suspect dependency on the TSC could be ameliorated in many cases where non-x86 platforms are desired. For example, 64-bit Linux can support a nanosecond resolution `gettimeofday()` *virtual system call* [31], which has less overhead than a regular system call and is specifically intended for use in real-time systems. This facility depends on the availability of userspace-accessible timers.

Besides decreasing portability, the TSC has some potential drawbacks [29, 31]. First, power management functionality (for example, dynamic voltage and frequency scaling features) can cause the TSC to be an unreliable source of time. Typically, such features are controlled by the operating system per the ACPI standard [2], and can be enabled or disabled by the user. Users should also check BIOS settings for power management options. Second, *system management interrupts* on some hardware can also render the TSC unreliable; thus, to use the library in a non-research system, hardware would have to be screened for this kind of interrupt. Finally, although Intel has pledged to support synchronization of the TSC across all cores on all upcoming platforms (as it is in our experimental platform), there is no guarantee of this in all current platforms [5].

Locking. The need for locks inside the user-level scheduler was discussed earlier. For the scheduler to perform well, these locks must be extremely performant. While the `pthreads` API provides for locks, it makes no guarantees about their underlying implementation. For example, on Linux, locks are realized using futexes [34], in which a kernel-level thread requesting a contended lock can be suspended by the kernel in favor of lower-priority work. Frequent suspensions of this kind could be devastating to the performance of the library.

Thus, the library implements *spin locks* (wherein contending threads wait on a shared variable) using architecture-specific assembly code. (Currently, only x86 is supported, but supporting other architectures is straightforward.) Spin locks are appropriate, since critical sections in the scheduler are extremely short.

Synchronization protocols. For short critical sections, real-time tasks can safely use the spin locks provided by the library (as described above) to protect critical sections.

The library also supports both short (spin-based) and long (suspension-based) resources under the FMLP, as mentioned in Section 2. This feature has been tested relatively well, and appears to have overheads commensurate with the rest of the library. (This is intuitively expected, because most of the FMLP implementation re-uses lower-level library features.) An empirical evaluation of FMLP-induced overheads was decided to be excessive for this paper.

Our goal was simply to show that the library can support global task-level dynamic-priority synchronization protocols; the choice of FMLP over other synchronization protocols was arbitrary.

Protecting against page faults. During initialization, the POSIX `mlockall()` function is used to ensure that memory is never swapped to disk, which is in accordance with established practice for hard real-time systems [9]. This guarantees that no page faults will occur while the library code itself is running. Any page fault that does occur is likely to be triggered from a proxy thread during a system call, and thus will not disrupt the scheduling of the real-time workload.

Idleness. During initialization, the library creates "idle tasks" that have the lowest possible priority, and therefore are scheduled when there is no other available real-time work. When selected to run, an idle task immediately calls the POSIX `sleep()` function, causing the host kernel-level thread to be suspended and allowing other (*i.e.*, background) work to be scheduled by the underlying RTOS. Whenever a signal is sent to the suspended thread, `sleep()` returns and

the idle task calls `schedule()`.

Standards compliance. A design goal of our library was to avoid esoteric operating system services that are not commonly present on UNIX-like operating systems.

The library makes use almost exclusively of POSIX-standardized features. Moreover, the features that are used are drawn from a relatively small set that revolves around thread creation and signals. We believe this set of features is likely to be well-supported, in general, even in RTOSs that are only partially POSIX compliant.

One non-POSIX feature used by our library is `pthread_setaffinity_np()`, which allows a kernel-level task to be pinned to a particular processor. We believe that this feature is nearly universally supported by multicore-capable RTOSs.

Another such feature is a set of portable functions for managing user-level threads: `makecontext()`, `swapcontext()`, and `getcontext()`. These functions were deprecated after the 2004 version of POSIX, but are still widely supported in practice, and nonetheless can be implemented relatively easily.

As mentioned before, one hardware-related portability barrier is use of the TSC.

The fast_swapcontext() function. The library's `fast_swapcontext()` function is called by `schedule()` to perform a context switch. It is an assembly-level replacement for the POSIX `swapcontext()` function. A replacement was desired for two reasons. First, `swapcontext()` includes an extra system call (thus introducing additional overhead), used to update the signal mask, that is superfluous for our purposes. Second, `swapcontext()` was deprecated after the 2004 edition of POSIX [16]. The drawback of providing a replacement is that such an approach introduces additional porting effort as the library is used on new machine architectures.[5] A similar `swapcontext()` replacement has been implemented elsewhere (for example, [40]).

Rather than provide a replacement for `swapcontext()`, it appears tempting to use the POSIX `setjmp()` and `longjmp()` functions. Unfortunately, POSIX forbids the use of these functions across different POSIX threads, which makes that approach unsafe for the library.

Context initialization. In order to initialize the contexts used by real-time tasks, our implementation currently relies on the POSIX `getcontext()` and `makecontext()` functions, which were also deprecated after 2004. However, contexts can be initialized in a fully portable and POSIX-compliant way, as demonstrated in [32], at the expense of additional implementation effort. That method may be adopted in a future release of the library.

Device interrupts. When a device raises an *interrupt*, the operating system suspends whatever work is being performed at the time, and executes a piece of code on behalf of the device. In order to decrease the impact of interrupts on scheduling, some RTOSs employ a technique known as *split interrupt handling*. Under this technique, each interrupt is split into a *top half*, which executes immediately when the interrupt is issued, and a *bottom half*, which is deferred. Bottom-halves are executed in kernel-level threads spawned by the kernel for that purpose. Such threads are assigned priorities that are appropriate for the device or devices they serve (which, in turn, depend on which real-time tasks use the services provided by those devices).

In the ideal case, our library would provide some mechanism to execute bottom halves in a dynamic-priority manner. Unfortunately, we are not aware of any portable technique that would make this possible.[6] Thus, our library leaves the scheduling of bottom halves to the underlying RTOS, if it supports split interrupt handling.

When split interrupt handling is available, care must be taken to ensure that bottom halves are executed in a timely manner. For each bottom-half processing thread, there are three options. First, the thread may be pinned to a core that does not host a worker thread, and assigned an appropriate static priority. Second, the thread may be assigned a static priority higher than the worker threads. In effect, this replicates non-split interrupt handling. Finally, if the thread's bottom halves may be treated as having lower priority than all of the library's real-time tasks, then the thread may be assigned a static priority lower than that of the worker thread.

Unfortunately, our library provides less flexibility with regard to bottom-half processing than would otherwise be available. In a fully static-priority system (*i.e.*, without our library), it would be possible to assign a bottom-half processing thread a priority in between that of two real-time application threads.

Other device interaction. In addition to interrupt handling, there are two other situations in which device driver code is executed.

The first case is when a real-time task issues a system call that causes device driver code to run in the kernel. This situation does not merit special concern for a user of our library. (However, real-time tasks must obey the protocol for making system calls that was described in Section 3.4.)

The second case is when device driver code (excluding interrupt handlers) resides in a kernel-level thread. For example, in Linux, the `kblockd` thread periodically executes driver code for block devices (such as disks) [24]. If the services provided by such threads are needed by real-time tasks, one must make appropriate provisions. The options available are identical to those described earlier for bottom-half processing threads.

Just as our library does not require any modifications to be made to the RTOS kernel, nor does it require any modifications to be made to code that interacts with devices.

4. Empirical Evaluation

In this section, we present overhead and latency measurements for the library. In general, these measurements fall in the range of ones to tens of microseconds. Under more taxing configurations (*i.e.*, larger C-EDF clusters), measurements range into the hundreds of microseconds.

5. This drawback is minor, because existing projects like `glibc` [4] provide largely equivalent code for many architectures.

6. There may be non-portable, operating system-specific techniques. For example, the QNX Neutrino RTOS allows bottom halves to be executed in userspace [17].

The remainder of this section is organized as follows. First, we specify the latencies and overheads that are of interest. Then, we describe our experimental methodology. After that, we present our measured results and analysis. Finally, we describe an experiment to validate the robustness of our implementation in which soft real-time guarantees were maintained over a 24-hour period.

4.1. Overheads and Latencies

Overheads and latencies relevant to our library are as follows. (The ordering here matches that used in Tables 1 and 2.)

Event latency. Event latency is the amount of time that elapses between the periodic release time of a real-time task and the corresponding invocation of the release handler.

Release overhead. Release overhead is the duration of execution of the release handler, minus the time taken to request preemption signals for remote processors. We were motivated to measure the latter separately from the former by the observation that they change independently with respect to various experimental parameters (as discussed immediately below).

Request overhead. Request overhead is the time taken, within the release handler, to request preemption signals for remote processors. There is no POSIX mechanism to request that multiple signals be sent with one system call; rather, our implementation invokes `pthread_kill()` repeatedly, in a loop (once per remote processor that needs to be preempted). In our experiments, this was observed to be one of the most significant sources of overhead.

Signal latency. Signal latency is the amount of time that elapses between a signal being requested by the release handler, and the corresponding invocation of the preemption handler on a remote processor.

Scheduling overhead. Scheduling overhead is the duration of `schedule()`, minus the time taken to perform a context switch.

Context switch overhead. Context switch overhead is the time taken to perform a context switch.

4.2. Experimental Methodology

In this subsection, we discuss our experimental methodology. We subdivide the discussion into the underlying platform, RTOS setup, and the experimental workload.

Underlying platform. An eight-core, 2.493-GHz Intel Xeon machine was used as the experimental hardware platform. Each core has private L1 instruction and data caches, and shares an L2 cache with a neighboring core. Power management features were disabled in the BIOS, and the machine was physically disconnected from the network during experiments. We believe that this platform is representative of hardware that manufacturers of advanced UAVs wish to deploy in upcoming systems (see Section 1).

The Linux kernel with the `PREEMPT_RT` patch (version 3.0.14-rt31) was used as the underlying RTOS. `PREEMPT_RT` is a major effort on the part of core Linux contributors to eliminate uninterruptible sections of the Linux kernel of long duration, among other changes geared towards enabling low-latency Linux deployments. It has been used in various industrial applications [13], and is also the basis for at least two commercial RTOS offerings [8, 12]. We chose to use `PREEMPT_RT` Linux over a commercial RTOS (such as VxWorks) because it is open source; this makes it more easily accessible to a wider range of researchers. In future work, we hope to examine the performance of our library across a variety of RTOSs.

RTOS setup. In order to ensure that TSC readings would not be perturbed by CPU frequency scaling or other power management features, the `PREEMPT_RT` kernel was compiled with all such features disabled.

By default, `PREEMPT_RT` processes interrupt bottom halves in a series of special-purpose kernel-level threads. To prevent the starvation of bottom halves, the system was configured such that the library's worker threads had a lower `SCHED_FIFO` priority than the `PREEMPT_RT` bottom-half-processing threads, yet a higher priority than all other kernel-level threads. The `chrt` command was used to re-assign priorities.

As a protection against faulty kernel-level threads that could starve the system, `PREEMPT_RT` has a feature that caps the utilization available to each thread. This feature was disabled by setting the `/proc/sys/kernel/sched_rt_runtime_us` parameter to -1.

Experimental workload. We tested four cluster configurations, as listed in Tables 1 and 2. We tested task systems comprised of $m \cdot k$ tasks, where k ranges from two to twenty in steps of two. For each configuration and each of the ten values of k, five task systems were produced, yielding 200 tasks systems in total. Each task system was executed for 30 seconds.

Our task systems were generated using the same methodology as in an earlier LITMUS$^{\mathrm{RT}}$ kernel-level implementation study [26]. Specifically, tasks were generated in groups wherein utilization is close to but not more than one; d such groups are assigned to a cluster of size d. Each group has one of the following randomly-chosen utilization distributions, as proposed by Baker [22]: light uniform, light bimodal, light exponential, medium uniform, and medium bimodal. Task periods fall uniformly in the range [$10ms$, $100ms$]. The execution time of each task is determined based on its assigned utilization and period.

Processors were allocated to clusters in accordance with the cache hierarchy of the machine; for example, a cluster of size two would be allocated processors sharing the same L2 cache. In addition, m non-real-time processes that repeatedly access large arrays in memory were used as background work. This ensures that cache lines used by the library are not permanently resident in cache memory, which could unfairly bias the overhead and latency measurements.

4.3. Results and Analysis

In total, 170,737,806 unique scheduling events were recorded. These events were then distilled into the average- and worst-case overheads and latencies presented in Tables 1 and 2.

	Event Latency	Release Overhead	Request Overhead	Signal Latency	Scheduling Overhead	Context Switch Overhead
$c = 8, d = 1$	10.81	3.07	0.08	–	0.37	0.51
$c = 4, d = 2$	11.25	4.01	4.07	13.38	1.25	0.89
$c = 2, d = 4$	11.59	4.71	10.43	40.57	1.57	1.18
$c = 1, d = 8$	13.18	6.02	52.62	62.84	1.63	1.29

Table 1: Average-case latencies and overheads, in μs.

	Event Latency	Release Overhead	Request Overhead	Signal Latency	Scheduling Overhead	Context Switch Overhead
$c = 8, d = 1$	84.60	32.30	7.06	–	21.72	18.31
$c = 4, d = 2$	86.29	31.38	65.94	91.23	18.27	20.84
$c = 2, d = 4$	78.24	27.53	84.20	120.80	19.00	18.78
$c = 1, d = 8$	60.64	36.53	218.61	187.13	32.58	18.90

Table 2: Worst-case latencies and overheads, in μs.

There is no reported signal latency for the $c = 8, d = 1$ case, because signal latency measures the time required to trigger remote preemptions, which are never used in a cluster of size one. For the same reason, no requests to send a remote preemption signal are made in this particular case. Counterintuitively, we report a worst-case request overhead of approximately 7 microseconds. This is likely due to the inopportune firing of hardware interrupts while the remote preemption signal requesting code was being skipped over.

The work most directly comparable to our library is LITMUS$^{\mathrm{RT}}$. As in our work, latencies and overheads under clustered schedulers in LITMUS$^{\mathrm{RT}}$ fall within the range of ones to tens of microseconds for small clusters, and sometimes hundreds of microseconds for larger clusters[7] [26]. However, in LITMUS$^{\mathrm{RT}}$, overheads and latencies tend to be distributed differently among the various measurements, which are not always directly comparable to our own. For example, LITMUS$^{\mathrm{RT}}$ does not accrue significant request overhead, but has greater context switch overhead; and LITMUS$^{\mathrm{RT}}$ tends to have smaller release overhead, but greater scheduling overhead. It would be interesting to find out if LITMUS$^{\mathrm{RT}}$ strictly outperforms our library (or vice versa), or, possibly, under which circumstances each one performs better. To conduct such a comparison would require incorporating measurements for each system into a full schedulability analysis (as described in [26]), and would require a more detailed treatment than can be provided in this paper, but is intended for future work.

One important observation is that with larger clusters, request overhead and signal latency begin to grow significantly. In general, LITMUS$^{\mathrm{RT}}$ studies have shown that small cluster sizes tend to dominate larger cluster sizes in terms of schedulability anyway; thus, this result is not greatly concerning.

4.4. Robustness Experiment

In order to ensure the robustness of our implementation, we conducted an experiment in which a single task system executed continuously for over 24 hours. Soft real-time correctness was ensured by a special "watchdog task" that executed once every second. This task ensured that all other tasks had completed the proper number of jobs up to that point within a small tolerance. The tolerance was necessary to accommodate the fact that the system was over-provisioned from a hard real-time perspective, guaranteeing the existence of some tardiness during certain intervals.

7. We refer to a LITMUS$^{\mathrm{RT}}$ study on a Xeon L7455 platform as opposed to the Xeon E5420 used in these experiments. The E5420 is marginally faster, among other minor differences.

5. Conclusion and Future Work

As the complexity and diversity of resource allocation techniques has grown, supporting them at the kernel level has become increasingly burdensome. Thus, we have proposed supporting these techniques at the user level. Our results are sufficient to demonstrate that a userspace library can support the class of resource allocation methods of interest, with latencies and overheads that are sufficiently small as to make our library relevant for a subset of future real-world real-time applications. While our library is currently a proof-of-concept research effort, we hope to see it become a platform for both broader research and practical deployment.

In addition to the future work mentioned previously, we plan to expand this research along a number of fronts. First, we would like to precisely characterize the scalability of the library as cluster size and the number of clusters increase, and investigate techniques to improve scalability, with an eye towards advancing capabilities for real-time systems on manycore platforms. Second, we would like to better characterize tradeoffs in terms of schedulability under various schedulers and locking protocols. Third, we would like to investigate supporting advanced techniques for future avionics systems, including adaptable scheduling and mixed criticality. Finally, we would like to investigate software engineering concerns. In particular, we would like to support virtual machines for high-level languages, which could provide an avenue for memory protection, among other advantages. In addition, we would like to develop "self-monitoring" schedulers that can provide data about their performance, and perhaps even their correctness.

We are very eager to receive feedback on the source code

of our library [11], which we anticipate maintaining as an open-source project.

Acknowledgments. We would like to thank the anonymous referees, each of whom provided useful advice that contributed to this paper. We would also like to thank Daniel M. Johnson, of Northrop Grumman Corp., and Glenn A. Elliott, of the University of North Carolina at Chapel Hill, both of whom also influenced this paper.

References

[1] ACE, TAO, and CIAO Success Stories. http://www.cs.wustl.edu/~schmidt/TAO-users.html.

[2] ACPI home page. http://acpi.info.

[3] Evidence web site. http://www.evidence.eu.com.

[4] GLIBC, the GNU C Library. http://www.gnu.org/software/libc/.

[5] Intel 64 and IA-32 Architectures Software Developer's Manual. Section 17.12.1.

[6] LITMUS$^{\mathrm{RT}}$ web site. http://www.cs.unc.edu/~anderson/litmus-rt/.

[7] Real-Time CORBA with TAO. http://www.cs.wustl.edu/~schmidt/TAO.html.

[8] RedHat MRG Realtime. http://www.redhat.com/products/mrg/realtime/.

[9] RT PREEMPT HOWTO. https://rt.wiki.kernel.org/articles/r/t/_/RT_PREEMPT_HOWTO_6bc9.html.

[10] SCHED_DEADLINE web site. http://gitorious.org/sched_deadline/pages/Home.

[11] Source code for the library described herein. http://cs.unc.edu/~mollison/userspace-library/.

[12] SUSE Linux Enterprise Real Time Extension. http://www.suse.com/products/realtime/.

[13] Systems based on Real time preempt Linux. https://rt.wiki.kernel.org/articles/s/y/s/Systems_based_on_Real_time_preempt_Linux_29a7.html.

[14] wuthreads: Implementing a user-space threading library. http://www.arl.wustl.edu/~fredk/Courses/OS/wuthreads.html.

[15] Xenomai web site. http://www.xenomai.org/.

[16] *IEEE Std 1003.1, 2004 Edition*, 2004. See entry on 'makecontext, swapcontext'.

[17] QNX Software Systems Limited. QNX Neutrino RTOS: System Architecture. 2012.

[18] A. Anantaraman, A. Mahmoud, R. Venkatesan, Y. Zhu, and F. Mueller. EDF-DVS Scheduling on the IBM Embedded PowerPC 405LP. In *Proceedings of the IBM P=ac^2 Conference*, 2004.

[19] A. Anantaraman, A. Mahmoud, R. Venkatesan, Y. Zhu, and F. Mueller. EDF-DVS Scheduling on the IBM Embedded PowerPC 405LP Progress Report. Technical report, 2004. http://moss.csc.ncsu.edu/~mueller/rt/rt04/g9/progress.pdf.

[20] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[21] M. Asberg, T. Nolte, S. Kato, and R. Rajkumar. ExSched: An External CPU Scheduler Framework for Real-Time Systems. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 240–249, 2012.

[22] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical Report TR-051101, 2005.

[23] A. Block, B. 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.

[24] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, 2006.

[25] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the ACM International Conference on Embedded Software*, pages 69–78, October 2011.

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

[27] J. Calandrino, D. Baumberger, T. Li, J. Young, and S. Hahn. LinSched: The Linux Scheduler Simulator. In *Proceedings of the ISCA 21st International Conference on Parallel and Distributed Computing and Communications Systems*, pages 171–176, September 2008.

[28] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, December 2006.

[29] J. Corbet. Counting on the time stamp counter. http://lwn.net/Articles/209101/, November 2006.

[30] L. Dozio and P. Mantegazza. Real time distributed control systems using RTAI. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 11–18, May 2003.

[31] J. Edge. The trouble with the TSC. http://lwn.net/Articles/388188/, May 2010.

[32] R. S. Engelschall. Portable multithreading: the signal stack trick for user-space thread creation. In *Proceedings of the USENIX Annual Technical Conference*, pages 20–20, 2000.

[33] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina. An implementation of the earliest deadline first algorithm in Linux. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1984–1989, 2009.

[34] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[35] C. Gill. *Flexible Scheduling in Middleware for Distributed Rate-Based Applications*. PhD thesis, Washington University, 2002.

[36] J. Lelli, D. Faggioli, and T. Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 6–15, July 2011.

[37] B. Lin and P. A. Dinda. VSched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of ACM/IEEE Supercomputing 2005*, 2005.

[38] M. Mollison and J. Anderson. Virtual real-time scheduling. In *Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–40, July 2011.

[39] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA - adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, 2009.

[40] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 376–387, 2010.

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

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

[43] V. Yodaiken. The RTLinux Manifesto. In *Proceedings of the 5th Linux Expo*, 1999.