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 demonstrate the first userspace scheduler that supports preemptive, dynamic-priority, migrating real-time tasks on multicore hardware, and show that it achieves latency and overhead properties approximately commensurate with a kernel-based approach. This result overturns conventional assumptions about real-time scheduler implementation, and offers a promising mechanism for deploying more effective multicore real-time systems.

1. Introduction

In recent years, significant research effort has been expended upon resource allocation methods for real-time workloads with the following basic properties: tasks execute on a multicore machine; tasks are fully preemptive; and task priorities can change dynamically at runtime.

In general, such properties permit scheduling algorithms and locking protocols to be developed with highly desirable characteristics. For example, recent work has shown that the *clustered earliest-deadline-first algorithm* (C-EDF) performs well on large multicore machines [21], and an asymptotically optimal locking protocol that can be used with this algorithm has been given [19].

The set of scheduling algorithms and locking protocols developed for these kinds of workloads has rapidly diversified, and these resource allocation methods have become rather complex. In many cases, researchers have focused on supporting increasingly more realistic and sophisticated workloads. For instance, there are now resource allocation methods to support tasks that suspend their execution [29]; those wherein the timing properties of tasks vary dynamically [9, 16]; and those wherein tasks of mixed *criticality* (assurance against failure) [14, 35] exist.

Contributing to the trend of increasing diversity and complexity, is the desire by some engineers to customize or embellish certain resource allocation methods to suit the needs of specific applications. For example, engineers developing software for unmanned aerial vehicles (UAVs) expressed to us the desire to convey domain-specific knowledge about the vehicle's current mission to the scheduler, in order to make better-informed scheduling decisions.

Problem Statement. In current practice, scheduling algorithms and locking protocols are supported in software at the level of the operating system (OS) kernel.¹ Unfortunately, this approach presents serious barriers to the adoption of the class of resource allocation techniques described above, for practical use in industry. Below, we list some of these barriers, which we have encountered in work with industry colleagues and also in work to support new resource allocation techniques at the kernel level.

- 1) *Robustness.* Adding complex functionality to the RTOS kernel scheduler is likely to introduce bugs. Oftentimes, these bugs are difficult to detect, as they may occur only under infrequent race conditions, or only under certain workloads. Moreover, they are often difficult to understand and correct. This makes it highly time consuming to develop new functionality. This problem is exacerbated as the needed degree of assurance against failure increases. See [20] and [21] for more information.
- 2) *Maintainability and Extensibility.* Modifying the kernel to support new resource allocation methods requires expert knowledge of both the particular kernel being modified, and the novel resource allocation methods being supported. Expertise in either of these areas is rare, and expertise in both is essentially nonexistent outside a small community of researchers. This presents a major problem for system developers who have to produce a working system on time and on budget, and maintain it over its lifespan—potentially many years.
- 3) *Portability.* Adding specialized functionality to a kernel ties users not only to that specific OS (e.g., Linux or VxWorks), but also to that specific kernel version. This is a serious problem if the kernel needs to be updated (e.g., due to security fixes or new functionality).
- 4) *Market Forces.* Waiting for RTOS providers (commer-

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.

1. There are some notable exceptions to this rule for *non-preemptive* tasks, as described in Section 2.

cial vendors and open source projects) to implement needed functionality is unlikely to pay off for many industry practitioners. Besides facing the problems mentioned above, providers are incentivized to supply a reliable platform for the majority of their users. This is often directly at odds with supporting emerging novel paradigms. Moreover, any selected subset of techniques supported in this manner is likely to prove insufficient as new use cases and techniques emerge over time, and necessarily will leave out practitioners who desire (potentially proprietary) customizations for particular niche use cases.

Given this situation, the search for an alternative for supporting novel resource allocation methods that does not require modification of the kernel is well-motivated. In fact, our work with industry colleagues—and our dismay that we could find no existing, practical method for them to deploy certain resource allocation techniques ideal to their application domain—has directly motivated this work. Regardless of this practical motivation, we consider the fundamental, underlying question—to what degree certain resource allocation techniques can be supported outside the kernel—to be inherently scientifically interesting.

Contributions. It seems to be a common assumption among real-time researchers that advanced scheduling algorithms and locking protocols supporting fully preemptible tasks can only be supported in the kernel while maintaining relatively small overheads, or can only be supported in the kernel regardless of overheads. In this paper, we overturn that assumption by showing that fully preemptible, dynamic-priority resource allocation mechanisms for tasks that migrate among processors can be supported by a userspace library with relatively small overheads (which we quantify in Section 4). Our empirical evaluation was conducted atop the `PREEMPT_RT` version of the Linux kernel; however, because the library’s interface with the underlying OS is almost entirely POSIX compliant, we believe it to be highly portable. Our source code is made available online [12].

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.

Earlier, in [30], we examined the possibility of creating a library such as that presented herein, and raised a number of questions that are satisfied in this paper.

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 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 re-assigned). 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.² 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. (The word “eligible” is used to exclude tasks that are waiting for a shared resource to become available.) C-EDF exhibits many of the key features of other migrating, dynamic-priority, fully-preemptible scheduling algorithms. In many cases, it is the basic foundation of these algorithms. Thus, as of the time of writing, our userspace scheduling library supports C-EDF, and it is the basis of our empirical measurements (Section 4).

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. These include the *multiprocessor priority-ceiling protocol* (MPCP) [32] and the *distributed priority-ceiling protocol* (DPCP) [33]. More recently, the *flexible multiprocessor locking protocol* (FMLP) [17], and the *$O(m)$ locking protocol family* (OMLP) [18,21] were designed for use with dynamic-priority scheduling algorithms such as those discussed in the previous subsection. We leave an empirical evaluation of our userspace library with respect to synchronization protocols for future work. However, the means by which these protocols can be supported in the library are described in Section 3.

2.4. Related Software

There is a large pre-existing ecosystem of resource allocation software. Below, we discuss the most closely related software, which we subdivide into kernel-based and

userspace-based approaches. Due to space constraints, we omit a discussion of microkernel-based approaches and real-time programming languages. We also omit non-real-time work on both cooperative scheduling between userspace and the kernel, and parallel threading libraries. These topics are discussed in our earlier paper [30].

Kernel-based. In general, commercial and open-source RTOSs do not support dynamic-priority scheduling, apart from various research projects that extend them. One notable exception is the ERIKA Enterprise RTOS [25], which supports partitioned EDF scheduling. A classification and summary of existing RTOSs can be found in [21].

Of these research projects, the one most closely related to our library is LITMUS^{RT}, a Linux patch that supports the same class of schedulers and locking protocols via a plugin infrastructure. Although LITMUS^{RT} could hypothetically be used to support certain real-world real-time workloads, that is not a focus of its developers, and practitioners seeking to to use it in this manner would face the pitfalls delimited in Section 1.

Another Linux kernel patch, SCHED_DEADLINE [34], 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. While the success of this effort now appears imminent, the longevity of an effort to mainline a single scheduling algorithm serves to reinforce the points made in Section 1.

Recent work has investigated “minimally invasive” techniques to provide novel resource allocation methods in the kernel without altering existing kernel code. Under these techniques, kernel data structures are either directly modified by scheduler code, or modified indirectly by calling internal scheduler-related functions. This limits the memory safety and portability of these techniques.

A primary example of this line of research is RESCH [28], which supports schedulers of the same class supported by our library, implemented as loadable Linux kernel modules. Unfortunately, RESCH has not been subjected to an empirical study of latencies and overheads. Another example of this approach is USR [15], which supports the implementation of single-core schedulers in VxWorks.

Userspace-based. There exists a wide body of work on *distributed, real-time, embedded* (DRE) middleware, which offers functionality beyond that provided by the underlying RTOS [27]. 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 [2]. This supports our hypothesis that a userspace approach holds promise for bringing the resource allocation techniques supported by our library to industry practitioners.

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

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

userspace scheduler [10,11]. That effort targeted only single-core machines; moreover, it did not aim to show applicability for real-world use. Thus, many of the complexities addressed in this paper did not arise. For example, no empirical evaluation of overheads and latencies was conducted, and portability (among other practical issues) was not discussed.

3. Implementation

The fundamental design and implementation problem of the library can be stated as follows. How can dynamic-priority, fully-preemptible, migrating tasks be supported, given that the underlying OS only offers static-priority tasks and a rather limited interface that can be used to control those tasks? Not any design will work: a successful design must exhibit sufficiently low overheads to be useful for at least some real-world applications. Moreover, it is important for the library to retain full control of the underlying processors (or as near to it as possible). These problems are addressed in this section.

The implementation of the library can be considered in terms of three distinct areas of infrastructure: realizing tasks, accessing time, and preemptive scheduling. In the subsequent subsections, we discuss each of these areas in turn. One additional subsection is devoted to miscellaneous other implementation issues.

Throughout the discussion, we devote special attention to pointing out any limitations imposed by the implementation, as well as discrepancies with regards to POSIX compliance and portability.

3.1. Realizing Tasks

From the question posed above, it can be concluded that there are two basic approaches for implementing a userspace real-time library. The first option is to manipulate the OS's task abstractions to cause them to behave in the desired manner, using system calls. The second option is to somehow realize or instantiate real-time tasks at a higher level, and multiplex these tasks atop the underlying OS task abstractions. Because the latter method seemed more likely to yield lesser runtime overheads, we selected that method for the implementation of our library. (However, we intend to explore the former method in future work.)

In order to realize real-time tasks atop underlying OS abstractions, the library initializes a three-level hierarchy of tasking abstractions prior to the release time of the first real-time task. This hierarchy is as follows.

Level 1. For each cluster, one process— c in total—is created. We use the term *process* to indicate a kernel-level entity that provides memory isolation from other processes. (A process contains one or more distinct kernel-schedulable entities, as discussed below.) In principle, these c processes can be created by the POSIX-provided `fork()` and `exec()` system calls. However, in our current implementation, these processes are created by a `bash` script that is part of a framework used to launch experimental task systems.

Level 2. For each cluster, d kernel-level threads—one for each processor in the cluster—are created by the corre-

sponding process. We use the term *kernel-level thread* to denote a schedulable entity known to the kernel. In this case, the kernel-level threads are created using the POSIX Threads (pthreads) API. We denote these threads as *worker threads*.

Each worker thread is statically assigned to its processor using the `pthread_setaffinity_np()`³ function, and is statically prioritized above all non-real-time work by assigning it a `SCHED_FIFO` priority using `pthread_setschedparam()`. This ensures that the worker thread has full use of the processor at all times, except when work must be performed by the kernel (for example, servicing a hardware interrupt).

Level 3. Also within each process, the library creates a user-level thread for each real-time task assigned to the cluster corresponding to that process. We use the term *user-level thread* to denote an entity unknown to the kernel, and scheduled within a kernel-level thread.

In our library, user-level threads are supported as follows. A separate call stack is allocated for each thread. To switch to a different thread, the processor context of the currently-running task (*i.e.*, the complete set of registers, including the stack pointer) is stored in memory, and the context being switched to is loaded from memory. More information about this technique is available in [24].

Limitations. Under this scheme, real-time tasks are provided memory protection from tasks assigned to other clusters, but not from those assigned to the same cluster. We presume this to be a fundamental limitation of any userspace real-time scheduling library wherein real-time tasks are realized at a level above the underlying OS tasking abstractions.

An approach to providing complete memory protection under such a scheme is to encapsulate each real-time task in a virtual machine hosted atop the corresponding user-level thread (for example, a Java Virtual Machine (JVM) interpreter). Besides isolating tasks from one another, this approach provides other well-known benefits associated with interpreted languages (such as programmer ease-of-use), at the expense of higher runtime overhead. We are eager to explore this option in future work.

We consider the lack of memory protection among all tasks to be the most serious limitation (by far) of the library as presented in this paper. Nonetheless, there are several considerations that we believe mitigate this issue.

First, in contrast to general-purpose systems, memory protection in real-time systems is often treated not as a requirement, but merely as a feature. This can largely be attributed to the fact that real-time systems tend to be subject to rigorous engineering practices, and to be deployed in a strictly controlled manner. Consider that VxWorks (perhaps the most widely-used RTOS) only supported memory protection in the 6.0 series (released in 2004).

Second, the library does support memory protection between tasks in separate clusters. Thus, if software compo-

3. POSIX does not offer an official mechanism to assign threads to processors. In our experience, such a mechanism is nearly universally supported on multiprocessor OSs. Perhaps this discrepancy is explained by the fact that the bulk of POSIX dates to an era before multiprocessor systems were common.

nents developed or acquired separately can be assigned to separate clusters, this protection is likely to be sufficient. (Fortuitously, the library would allow each such component to implement a separate, custom-tailored resource allocation policy.)

Finally, sacrificing memory protection among tasks, in return for memory protection between the kernel and scheduler code, may be worthwhile if the scheduler is sufficiently complex. For example, under our library, a fault in the scheduler in one cluster would bring down only that cluster (not the entire system); the same cannot be said of schedulers implemented inside the kernel. Scheduler code is a prime example of inherently tricky code that is prone to all the perils of multithreaded programming (particularly, deadlocks and race conditions). In contrast, real-time tasks are (typically) implemented with sequential, relatively straightforward code.

3.2. Accessing Time

In developing our library, we made the assumption that a low-overhead, high-resolution mechanism to access time is needed.⁴ To achieve this, the 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.⁵ We use the TSC to produce a detailed record of runtime events, stored in memory, that can be analyzed offline to ensure that the scheduler was implemented correctly and to calculate overheads (such as those provided in Section 4). We also use the TSC throughout the code that drives enforcement of the sporadic task model—for example, to ensure that when a job completes, another job of the same task becomes eligible to run only if the task’s period has elapsed.

Limitations. Unfortunately, because the TSC is an architecture-specific feature, it is not mandated by POSIX, and the portability of our implementation across hardware platforms is constrained. (Since the TSC can be read directly from userspace, portability across OSs is not affected.)

Even given an x86 platform, there are some major caveats to using the TSC for accurate timekeeping [22, 23]. Power management functionality (for example, dynamic voltage and frequency scaling features) can cause the TSC to be an unreliable source of time; thus, our library presumes that these features have been disabled at the kernel level. In addition, *system management interrupts* on some hardware can also render the TSC unreliable; thus, we assume that for hard real-time systems, hardware has been vetted for this kind of interrupt. Finally, synchronization of the TSC across cores—assumed by our current implementation—is not guaranteed on all current platforms, though Intel has pledged to support it universally in the future [4].

Like the memory protection issue discussed earlier, we see the limitations of the TSC as being far from ideal, but not necessarily problematic. Hardware dependencies are

fundamental to real-time systems; engineers must carefully select and understand the platform upon which they wish to deploy. Moreover, there are many hardware platforms to choose from. We thus consider portability of a userspace real-time library across different hardware platforms to be secondary in importance to portability across different OSs.

Nonetheless, in future work, we will investigate ways to ameliorate dependence on the TSC. For example, 64-bit Linux can support a nanosecond resolution `gettimeofday()` *virtual system call* [23] which has less overhead than a regular system call. Moreover, in the case that fine-grained tracing information is not needed, we suspect that it may be possible to account for time based on the expiry of POSIX timers that are already used for triggering preemptions, as discussed in the next subsection.

3.3. Preemptive Scheduling

In the previous two subsections, we have demonstrated the realization in userspace of two areas of functionality normally reserved to the kernel: a task abstraction, and low-overhead access to time. We now turn our attention to a method by which tasks can be preemptively scheduled at runtime, allowing our library to maintain the invariant that within each cluster, the d highest-priority eligible tasks are scheduled. The need for preemptivity arises because a newly-released task may be among the d highest-priority tasks.

Signals and Timers. The key mechanism that enables preemptivity to be realized in userspace is the POSIX signaling functionality. POSIX defines a range of numerically-identified *signals*. A system call is used to send a signal to a kernel-level thread (either an arbitrary thread in a specific process, or a specific thread).

When a signal is sent, the kernel refers to the *signal mask* of the process to determine which kernel-level threads (if any) have the signal *unblocked* and, thus, can receive the signal. If no thread can receive the signal, it remains *pending* until unblocked by one of the threads. While a signal is pending, an additional signal of the same type sent to the same destination will be dropped.

When a signal is delivered, the kernel refers to the *signal table* of the process. This table associates signals with *signal handler* functions. The kernel interfaces with the C runtime library to asynchronously interrupt the kernel-level thread by invoking the specified signal handler. (If none is specified, a default handler for the signal will be used.) Note that the signal table can be adjusted by any kernel-level thread in the process, while each kernel-level thread controls its own part of the signal mask.

After the signal handler returns, execution in the kernel-level thread continues from the point where it was interrupted.

In addition to signaling, POSIX provides methods to create timers and arm them. When a timer fires, a user-specified signal is sent to the process that originated the timer.

Triggering Task Releases. During the initialization phase, the library creates a POSIX timer for each cluster that is

4. Accessing the time is distinct from arming a timer (discussed subsequently).

5. On a processor with clock speed of more than 1 GHz, this yields timing resolution of better than one nanosecond.

used to trigger the release of any periodic real-time tasks in the task system. Throughout runtime, this timer is re-armed as necessary in order to cause it to fire at the time of the next periodic task release. When the timer fires, the kernel sends a signal that causes the library-provided `release_handler()` function (henceforth, the “release handler”) to be invoked in an arbitrary kernel-level thread in the cluster.

The release handler always transfers all (newly) eligible tasks in the release queue, to the ready queue. (This necessitates a re-arming of the release timer if any tasks remain in the release queue.)

In addition, the release handler must trigger scheduling on any processors in the cluster that are executing lower-priority work than any newly-released task. To achieve this, the release handler sends a library-specified signal (henceforth, the “preemption signal”) to any processors that meet this criterion, other than the one upon which it is running; their response to receiving this signal is discussed below. Then, if the release handler is running upon a processor that meets this criterion, it synchronously invokes the library’s `schedule()` function (also discussed below). Otherwise, it simply returns.

Triggering Remote Rescheduling. Receipt of the preemption signal by a kernel-level thread causes the library-provided `preemption_handler()` function (henceforth, “preemption handler”) to be invoked. The preemption handler simply calls `schedule()`; when `schedule()` returns, it returns.

The Schedule() Function. If the task at the head of the ready queue has higher priority than the currently-scheduled task on the processor, `schedule()` causes a context switch to the task at the head of the ready queue. Before the switch, `schedule()` removes that task from the ready queue, and adds the currently-running task to the ready queue. The switch is actually performed by calling the `fast_swapcontext()` function (discussed below).

As described above, the `schedule()` function can be triggered asynchronously (relative to a running task) by being called from a signal handler (either the release handler or the preemption handler). In such a case, the signal handler returns only after the task it interrupted is selected to run again from the ready queue in some subsequent invocation of `schedule()`.

Note that there is one additional (synchronous) situation when `schedule()` is called: when a job completes. In that case, the currently-running task is not necessarily added to the ready queue before the context switch; instead, if its period has not elapsed, it is added to the release queue (which may necessitate re-arming the release timer).

Limitations. The features described in this subsection are present in POSIX; however, there are two areas of concern regarding temporal correctness.

First, POSIX does not specify a worst-case timer resolution. Fortunately, this is typically guaranteed at the RTOS and hardware level.

Second, POSIX does not mandate a worst-case latency between a signal being requested, and the signal being

delivered. We believe that it is reasonable to expect the RTOS to support relatively low-latency signals in the case where at most a few signals are sent at once. However, our library may request that many signals be sent in a short amount of time. This raises questions about the scalability of the underlying signal-delivery infrastructure.

In our experimental evaluation, latencies due to timer and signal performance were generally on the order of tens of microseconds. We consider this result to be highly encouraging. More detailed information on this topic is presented in Section 4.

3.4. Miscellaneous Issues

In this subsection, miscellaneous implementation issues are discussed.

Locking. Because the scheduler can be invoked simultaneously on multiple processors, key data structures used by the library need to be protected by a lock. While the `pthread`s API provides for locks, it makes no guarantees about their underlying implementation. For example, on Linux, locks are realized using `futexes` [26], in which a kernel-level thread requesting a contended lock can be suspended by the kernel in favor of lower-priority work.

Frequent suspensions could be devastating to the performance of the library. Moreover, critical sections in the library code (*i.e.*, `schedule()`, `release_handler()`, `preemption_handler()`, and the functions they call) are extremely short. 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 trivial.)

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 [6]. This guarantees that no page faults will occur while the library code itself is running. If a page fault were to occur in a critical section of the library code, it could halt progress for every kernel-level thread in the given cluster, and thus for every real-time task in that cluster.

Sharing Resources Among Tasks. Real-time tasks can use the spin locks provided by the library (as described above) to protect critical sections. In future work, the library will be extended to support novel locking protocols. This will include support for user-level suspensions of real-time tasks, which can be achieved by using context switching techniques in a manner largely identical to that already employed by `schedule()`.

Blocking Signals in Critical Sections. We have seen that the library relies heavily on sending signals between processors to trigger preemptions. If a critical section of the library were to be interrupted by one of these signals, it would quickly result in deadlock (among other problems). Because of the completely asynchronous nature of signal delivery and signal handler invocation, overcoming this challenge is nontrivial.

A way around the problem is provided by a POSIX feature that allows the programmer to specify signals to be automatically blocked by the receiving thread when a particular signal handler is invoked, and automatically unblocked when it returns. Both of the library’s signal handlers are configured to behave this way with respect to one another’s signals, as well as their own signals.

However, this appears to present a new problem. Recall that `schedule()` can be called within a signal handler, allowing other tasks to run before the task that was preempted is resumed and the signal handler returns. This would seem to lead to an arbitrarily long span of time in which a signal handler is outstanding and signals are blocked, disabling preemptivity for that kernel-level thread.

In fact, signals are *not* blocked during that span. There are two cases to consider. First, if the job that is selected to run has already been preempted, as soon as it resumes, it will itself return from a signal handler (thus, automatically unblocking signals again). Second, if the job that is selected to run has not already been preempted, it therefore has not yet started running. To resolve this case, we require that tasks unblock signals at the beginning of every job.⁶

We have shown that the two signal handlers are protected from being interrupted by signals. From this, it is evident that the critical section in `schedule()` is protected when `schedule()` is called within a signal handler. However, `schedule()` can also be called at the end of a completing job. Thus, we require that tasks block signals at the end of any job, just before calling `schedule()`. This is safe, because we have shown that in every case, the next job that runs will immediately unblock signals again.

Performing 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 [1]. Because of the continual use of signals by the library, these situations are potentially problematic. Thus, we require that a task block the signals used by the library before invoking a system call, and then unblock them after the system call completes.

In the current implementation, if blocking occurs during a system call, the kernel-level thread will be suspended by the OS, making the processor it is assigned to temporarily unavailable for real-time tasks. From an analytic perspective, this is not necessarily problematic, as many (though not all) real-time locking protocols use *suspension-oblivious analysis* [18, 21] wherein time during which tasks are suspended is not recovered. However, depending on the workload, this behavior could be problematic from a performance perspective. In future work, we hope to rectify this situation by allowing a “proxy thread” to take the place of the kernel-level thread that is suspended and continue executing real-time tasks. (Many details must be accounted for in order to perform this operation correctly, as discussed

earlier in [30].)

Idleness. During initialization, the library creates d artificial “idle tasks” in each cluster. These tasks have the lowest possible priority; they are selected to run in a kernel-level thread by `schedule()` when there is no real-time work available. These tasks simply call the POSIX `sleep()` function, causing that particular kernel-level thread to be suspended and allowing other (*i.e.*, background) work to be scheduled by the system. Whenever a signal is sent to the suspended thread, `sleep()` returns and the idle task calls `schedule()`.

An alternate idle behavior was tested, in which the call to `sleep()` was replaced with a busy loop. This results in slightly lower overheads, but prevents the OS from scheduling background work.

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 [1]. The drawback of providing a replacement is that such an approach introduces additional porting effort as the library is used on new machine architectures.⁷ A similar `swapcontext()` replacement has been implemented elsewhere (for example, [31]).

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 inapplicable 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 [24], at the expense of additional implementation effort. That method will be adopted in a future release of the library.

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. Although a detailed comparison is beyond the scope of this paper, prior implementation studies of C-EDF at the kernel level [21] yielded approximately commensurate results. These prior studies have shown that when fully accounting for overheads and latencies, C-EDF can support a wide variety of real-time workloads.

6. Cosmetically speaking, this detail could be hidden from the implementor of a task. It is also possible for `schedule()` itself to perform this step, but doing so does not serve to simplify the library implementation.

7. This drawback is minor, because existing projects like glibc [3] provide largely equivalent code for many architectures.

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. 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 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 subsequently).

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 `sched_yield()`, 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 hardware platform used, the RTOS used, and the experimental workload.

Hardware 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).

RTOS. 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 is known to be used in various industrial applications [8], and is also the basis for at least two commercial RTOS offerings [5, 7]. 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.

The kernel was compiled with power management features disabled. In addition, the following two configuration steps were performed as the `root` user. First, permission for real-time threads to fully utilize the processor was granted by setting the `/proc/sys/kernel/sched_rt_runtime_us` parameter to 1. Second, the `ksoftirqd` kernel-level threads, which are used to offload certain work from the kernel, were assigned a higher `SCHED_FIFO` priority than that used by the library. If this step is omitted, the `ksoftirqd` threads are starved by the library’s kernel-level threads, resulting in the release timer interrupt signals not being delivered.

Experimental Workload. We tested four cluster configurations, as listed in Tables 1 and 2. We tested task systems comprised of $m \times k$ 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 task 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^{RT} kernel-level implementation study [21]. 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 [13]: 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. Measured Results

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 Table 1 and Table 2.

The absence of any signal latency for the $c = 8, d = 1$ case is explained by the fact that no preemption signals are

	Event Latency	Release Overhead	Request Overhead	Signal Latency	Scheduling Overhead	Context Switch Overhead
$c = 8, d = 1$	10.81 - 0.03N	3.07	0.08	–	0.37	0.51
$c = 4, d = 2$	11.25 - 0.01N	4.01	4.07 - 0.01N	13.38 - 0.04N	1.25	0.89
$c = 2, d = 4$	11.59	4.71	10.43 + 0.03N	40.57 + 0.03N	1.57	1.18
$c = 1, d = 8$	13.18	6.02 + 0.02N	52.62 + 0.21N	62.84 - 0.04N	1.63	1.29

Table 1: Average-case latencies and overheads, us ; N is the number of tasks.

	Event Latency	Release Overhead	Request Overhead	Signal Latency	Scheduling Overhead	Context Switch Overhead
$c = 8, d = 1$	84.60 - 0.02N	32.30 - 0.04N	7.06 - 0.03N	–	21.72 - 0.08N	18.31 - 0.05N
$c = 4, d = 2$	86.29 + 0.03N	31.38 - 0.03N	65.94 - 0.02N	91.23 - 0.03N	18.27	20.84 - 0.05N
$c = 2, d = 4$	78.24	27.53 + 0.02N	84.20 - 0.05N	120.80 - 0.04N	19.00 + 0.03N	18.78 - 0.04N
$c = 1, d = 8$	60.64 + 0.12N	36.53 + 0.15N	218.61 - 0.05N	187.13 - 0.06N	32.58 + 0.16N	18.90 - 0.03N

Table 2: Worse-case latencies and overheads, in us ; N is the number of tasks.

used: preemptions are always local with a cluster size of one. Request overhead is recorded, however, simply because the trace points placed on either side of the signal-requesting loop (the body of which would not be executed) remained present. The worst-case request overhead of approximately 7 microseconds is likely due to inopportune occurrences of hardware interrupts.

One important observation is that with larger clusters, request overhead and signal latency begin to grow significantly. In general, LITMUS^{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.

A cursory comparison of these measurements with similar measurements from prior LITMUS^{RT} studies suggests overall overheads and latencies are roughly comparable, but are distributed differently among the various measurements. For example, LITMUS^{RT} does not accrue request overhead, but has greater context switch overhead; and LITMUS^{RT} tends to have smaller release overhead, but greater scheduling overhead. These generalizations are relatively crude; it is impossible to be more specific without carefully taking into account many factors that differ between our library and LITMUS^{RT}, which is beyond the scope of this paper.

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.

4.5. Summary

We consider our results sufficient to declare that a userspace library can support the class of resource allocation methods of interest, with overheads and latencies approximating those found in a kernel-based technique. However, we cannot presently make a stronger or more specific claim. Due to space constraints, we reserve a much more thorough analysis of our results, and comparison to kernel-based approaches, for future work.

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. Moreover, we have shown that a relatively large class of techniques can be supported at the user level robustly and with latencies and overheads generally in the range of ones to tens of microseconds. 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.

We plan to expand this research along a number of important fronts. First, we would like to investigate a number of low-level implementation concerns. For example, how scalable is the library (both to more clusters and larger clusters), and how can scalability be improved? What are the greatest sources of interference from the kernel, and how can we design the library to mitigate these? Addressing the latter question will require an integrated tracing approach that

involves tools and data at the user and kernel levels. Second, we would like support several novel scheduling paradigms that have been the subject of recent research, including adaptive scheduling and mixed criticality scheduling, which we believe can play an important role in real-world systems in the near future. Many of these techniques have been the subject of past kernel-based implementation studies; we would like to extend prior work to more fully bridge the gap between theory and practical deployment. Finally, we would like to use the library as a basis to explore some more esoteric research ideas. For example, we would like to investigate whether a “self-checking” scheduler, which validates its own behavior and measures its own performance, can be created.

References

- [1] *IEEE Std 1003.1, 2004 Edition*, 2004. See entry on ‘make-context, swapcontext’.
- [2] ACE, TAO, and CIAO Success Stories. <http://www.cs.wustl.edu/~schmidt/TAO-users.html>, March 2012.
- [3] GLIBC, the GNU C Library. <http://www.gnu.org/software/libc/>, March 2012.
- [4] Intel 64 and IA-32 Architectures Software Developer’s Manual, March 2012. Section 17.12.1.
- [5] RedHat MRG Realtime. <http://www.redhat.com/products/mrg/realtime/>, March 2012.
- [6] RT PREEMPT HOWTO. https://rt.wiki.kernel.org/articles/r/t/_RT_PREEMPT_HOWTO_6bc9.html, March 2012.
- [7] SUSE Linux Enterprise Real Time Extension. <http://www.suse.com/products/realtime/>, March 2012.
- [8] 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, March 2012.
- [9] T. Abdelzاهر, E. Atkins, and K. Shin. Qos negotiation in real-time systems and its application to automated flight control. *IEEE Trans. Comput.*, 49(11), 2000.
- [10] 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=ac² Conference*, 2004.
- [11] 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. Available: <http://moss.csc.ncsu.edu/~mueller/rt04/g9/progress.pdf>.
- [12] Authors’ Web site. <http://www.cs.unc.edu/~mollison/>.
- [13] T. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. Technical Report TR-051101, 2005.
- [14] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, November 2011.
- [15] M. Behnam, T. Nolte, I. Shin, Å. Åsberg, and R. J. Bril. Towards Hierarchical Scheduling on top of VxWorks. In *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 63–72, July 2008.
- [16] A. Block. *Multiprocessor Adaptive Real-Time Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2008.
- [17] 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.
- [18] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 49–60, 2010.
- [19] 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.
- [20] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 157–169, December 2008.
- [21] B. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.
- [22] J. Corbet. Counting on the time stamp counter. <http://lwn.net/Articles/209101/>, November 2006.
- [23] J. Edge. The trouble with the tsc. <http://lwn.net/Articles/388188/>, May 2010.
- [24] 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.
- [25] Evidence Web site. <http://www.evidence.eu.com>.
- [26] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwoks: Fast userlevel locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.
- [27] C. Gill. *Flexible Scheduling in Middleware for Distributed Rate-Based Applications*. PhD thesis, Washington University, 2002.
- [28] S. Kato, R. Rajkumar, and Y. Ishikawa. A loadable real-time scheduler suite for multicore platforms. Technical report, 2010. Available: <http://www.contrib.andrew.cmu.edu/~shinpei/papers/rtsa10.pdf>.
- [29] C. Lui and J. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 425–436, December 2009.
- [30] 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.
- [31] 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.
- [32] R. Rajkumar. Real-time synchronization protocols for shared-memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.
- [33] 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.
- [34] SCHED_DEADLINE Web site. http://gitious.org/sched_deadline/pages/Home.
- [35] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 239–243, December 2007.