In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 25–36, Cancun, Mexico, December 3–5 2003.

Evolving real-time systems using hierarchical scheduling and concurrency analysis

John Regehr Alastair Reid Kirk Webb Michael Parker Jay Lepreau
School of Computing, University of Utah

Abstract

We have developed a new way to look at real-time and embedded software: as a collection of execution environments created by a hierarchy of schedulers. Common schedulers include those that run interrupts, bottom-half handlers, threads, and events. We have created algorithms for deriving response times, scheduling overheads, and blocking terms for tasks in systems containing multiple execution environments. We have also created task scheduler logic, a formalism that permits checking systems for race conditions and other errors. Concurrency analysis of low-level software is challenging because there are typically several kinds of locks, such as thread mutexes and disabling interrupts, and groups of cooperating tasks may need to acquire some, all, or none of the available types of locks to create correct software. Our high-level goal is to create systems that are evolvable: they are easier to modify in response to changing requirements than are systems created using traditional techniques. We have applied our approach to two case studies in evolving software for networked sensor nodes.

1. Introduction

By focusing on a single abstraction for concurrency, such as events or threads, traditional models for real-time and embedded software ignore a great deal of the richness and complexity that is present in actual systems. For example, in a typical embedded system based on an off-the-shelf real-time operating system (RTOS),

- interrupts are prioritized by a preemptive scheduler that is implemented in hardware,
- bottom-half handlers (a.k.a. tasklets, software interrupts, or deferred procedure calls) are scheduled non-preemptively in software,
- threads are scheduled preemptively by the RTOS, and
- lightweight events are run by schedulers implemented within individual OS-supported threads.

Each of these schedulers creates a unique *execution environment* — a context for running code that has particular performance characteristics and restrictions on actions that can be taken inside it. Furthermore, each pair of interacting execution environments has rules that need to be followed to implement mutual exclusion between tasks running in them.

Multiple execution environments exist because developers can exploit their diverse properties by mapping each task to an appropriate environment. For example, interrupt handlers incur very little overhead but provide an inconvenient programming model: blocking is not permitted and long-running interrupts unnecessarily delay other tasks. Threads, on the other hand, are more flexible and can block, but incur more time and space overhead.

The concurrency structure of a system is largely determined by the execution environments that it supports and by the way that tasks are mapped to these environments. For example, consider a pair of cooperating tasks. If both tasks are mapped to an event-driven execution environment then the tasks are mutually atomic — they run to completion with respect to each other, regardless of whatever preemption occurs in other parts of the system. On the other hand, if the tasks are mapped to separate threads, then the tasks can potentially preempt the other; they must each acquire a mutex before accessing any shared resources. Finally, if one task is mapped to a thread and the other is mapped to an interrupt handler, then the preemption relation is asymmetrical — the interrupt can preempt the thread, but not vice versa. In this case, the thread must disable interrupts when accessing a resource shared by the two tasks, but the interrupt does not need to take any special action.

Systems tend to evolve over time as features and requirements accumulate, and as the underlying platform changes to accommodate newer and better hardware. Also, all too often systems are designed in an exploratory way where code is initially structured in a simple, apparently adequate way, and then it is generalized several times before an en-

tirely working system can be created. As systems evolve it is commonplace for a task to be moved into a new execution environment. However, small changes such as moving a task from an event into its own preemptive thread, or from a thread into an interrupt, can easily break a previously correct system. For example, if code that is moved to interrupt context accesses a blocking lock, the new system risks crashing because it is illegal to block in interrupt mode. It is often far from obvious whether it is possible for a given piece of code to transitively call a blocking function. Similarly, if code that is moved from an event to a thread shares a resource with another event that had been run by the same scheduler, then the new system contains a potential race condition. These problems are often difficult to detect: a small change to the structure of execution environments in system, which might be implemented by changing a few lines of code, suddenly requires that the concurrency assumptions made by many thousands of lines of code be reexamined and modified.

This paper makes three main contributions to the engineering and analysis of real-time systems. First, in Section 2, we elevate execution environments to a first-class concept that can be used to better understand and validate real-time software. Previous work on real-time systems focused on a single execution environment, usually one based on preemptive multithreading — other environments, if present, were distinctly second-class. We present a framework for making generalized adjustments to the mapping of tasks to execution environments in order to help tasks meet deadlines. Our emphasis is on using real-time techniques to guide system design, as opposed to traditional real-time analysis that returns a binary result about schedulability. In particular, we are trying to create *evolvable* systems: those that are flexible and can gracefully adapt when faced with the changes in requirements that are inevitable as time passes.

Our second contribution, in Section 3, is to develop analyses for collections of execution environments. We show that it is feasible to compute response times, dispatching overheads, and blocking terms for tasks in real-time systems that are based on a hierarchy of preemptive and non-preemptive priority schedulers. We have also developed task scheduler logic (TSL): a formalism that can be used to derive and check rules for concurrency both within and across execution environments.

Our third contribution, in Section 4, is an empirical evaluation of our approach: we use it to solve two problems in meeting real-time deadlines in software for networked sensor nodes.

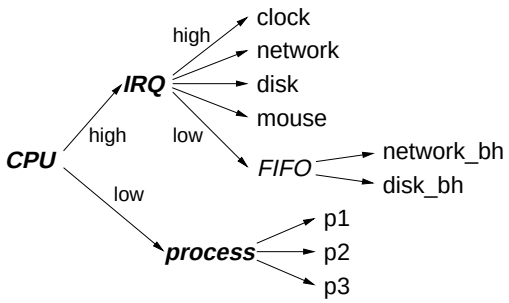

Figure 1. Scheduling hierarchy for a UNIX-like operating system. Clock, network, etc. are interrupt handlers, network_bh and disk_bh are bottom-half handlers, and p1–p3 are processes.

2. Hierarchical execution environments

Most real-time and embedded software is structured as a collection of execution environments. In this section, we explain this point of view and describe why it is useful. Our definition of *execution environment* is informal — it is the set of properties created by an instance of a scheduler and its global context that are of interest to developers.

2.1. An example

Because its structure is familiar to most readers, we use a generic UNIX-like scheduling hierarchy as an example; it is depicted in Figure 1. In this and subsequent figures, preemptive schedulers are in a bold, oblique font, non-preemptive schedulers in an oblique font, and non-scheduler entities in the hierarchy in the standard font. At the top of the UNIX hierarchy, the root scheduler, CPU, runs interrupts whenever possible; at all other times it runs a process scheduler in its user-mode context. Interrupts are prioritized by the IRQ scheduler that is implemented in hardware, with a software interrupt running at the lowest priority. The software interrupt handler contains its own scheduler that uses a FIFO scheduling policy: it runs bottom-half handlers to completion. The scheduling hierarchies found in many real-time operating systems are nearly identical to the one shown here.

2.2. Why execution environments?

Viewing a system as a hierarchy of execution environments is useful for several reasons. First, it facilitates understanding and reasoning about scheduling and concurrency relations. Second, we have found the hierarchical scheduling view of a system to be a useful and general notation — it can succinctly describe many different system architectures

and helps focus attention on the important similarities and differences between systems. For example:

- RTLinux [20] virtualizes the CPU's interrupt controller, adding a new root and a new thread scheduler to the scheduling hierarchy.
- TinyOS [9] contains execution environments equivalent to the bottom half of a traditional operating system, lacking a preemptive thread scheduler.
- Many applications extend the scheduling hierarchy by running new schedulers in a process or thread context. For example, an event-driven web server adds a new non-preemptive scheduler to the hierarchy, and a scientific application might add a preemptive user-level thread scheduler. Also, UNIX processes have an implicit signal scheduler that can preempt the main flow of control.

The diversity of execution environments in UNIX that Figure 1 illustrates is of little consequence to the majority of programmers who only write code that runs in a process context. However, people developing embedded and real-time applications frequently have the opportunity to write code for all available execution environments in order to create code that meets timeliness and efficiency goals.

The hierarchies discussed in this paper are superficially similar to our previous work on hierarchical CPU scheduling in HLS [15]: both are concerned with modular reasoning about compositions of schedulers. However, the work described in this paper differs in that it addresses heterogeneous mixes of execution environments and safety issues such as race conditions. The previous work, on the other hand, focused on guarantees about the availability of CPU time using reservation-like semantics. It did not address concurrency issues and was concerned only with a preemptive multithreading execution environment. Another significant difference is that while HLS specified a uniform interface for connecting schedulers, our current work is aimed at describing hierarchical schedulers as they "naturally" occur in existing systems. In practice, interfaces between schedulers tend to be non-uniform, are somewhat blurred by attempts to optimize systems, and may even be implemented in hardware.

2.3. Properties of hierarchical schedulers

To reason about hierarchical schedulers, we require that scheduler and task behavior is restricted in certain ways. The scheduling operations performed by a task are limited to:

1. Taking or releasing a lock.
2. Relinquishing control to its parent. This encompasses termination of a task instance as well as voluntary blocking.
3. Releasing a task that is run by a scheduler somewhere else in the hierarchy. For example, the clock interrupt handler in an RTOS might release a thread that is blocked waiting for a timer to expire.
4. Parameterizing schedulers; for example, a task requesting that its priority be raised.

A task that acts as a scheduler has access to two additional operations:

1. Dispatching a child task.
2. Suspending the currently running child task and saving its state.

We are working towards formalizing these restrictions; for now they remain intuitive. Real systems conform to them without modification, in our experience. However, our model cannot yet cope with exotic inter-scheduler protocols such as scheduler activations [1].

With a single exception, schedulers only receive control of the CPU from their parent or from one of their children. The exception makes preemption possible: control can be taken away from the currently running task and given to the root scheduler by the arrival of an interrupt. The root scheduler can then pass control down the hierarchy; if any scheduler along the way takes the opportunity to change its scheduling decision, it must save the context of its previously running task before it can dispatch a new one. Obviously, only preemptive schedulers can do this — non-preemptive schedulers must always wait for their currently running task to return control before making another scheduling decision.

To see how this works in practice, assume there is a machine that is running the hierarchy in Figure 1, that process p1 is currently running, and p2 is blocked awaiting data from the network. When a network interrupt arrives, p1 stops running and the root scheduler gives control to the IRQ scheduler, which saves a minimal amount of machine state and then dispatches the network interrupt handler. The network interrupt handler is designed to run very quickly; it acknowledges the interrupt and then releases the network bottom-half handler by first adding an element to the queue of tasks to be run by the FIFO scheduler and second, posting a software interrupt. When the network interrupt returns control to its parent, the IRQ scheduler finds that there is another interrupt pending — the software interrupt. The IRQ scheduler dispatches the FIFO scheduler, which dequeues and dispatches the network bottom-half handler. Now, assume that the packet being processed by the bottom-half handler is one that the blocked process p2 is waiting for,

and so the network bottom-half handler releases task p2 by calling into the process scheduler. After the software interrupt returns, the CPU scheduler gives control to the process scheduler which, instead of running p1, saves the state of p1 and dispatches the now-ready process p2.

In a system with hierarchical schedulers and execution environments, each lock is provided by a particular scheduler. Locks have the effect of preventing a scheduler from running certain tasks, effectively blocking sub-trees of the scheduling hierarchy. Therefore, taking a lock higher in the hierarchy potentially blocks more tasks than does a lock lower in the hierarchy. So, although root-level locks (like disabling interrupts) are often very cheap, holding them for too long can easily cause unrelated tasks to miss deadlines. This is an instance of the fundamental tension in real-time systems between creating code that makes efficient use of resources and code that reliably meets real-time deadlines.

2.4. Heuristics for evolving systems

A basic premise of this paper is that as systems evolve, it is sometimes necessary to restructure them so that tasks can meet real-time deadlines. Furthermore, we claim that the mapping of tasks to execution environments can usually be modified more easily than can other aspects of a system such as its algorithms, requirements, or underlying hardware.

As a first approximation, problems in meeting real-time deadlines can be caused by *transient overload*, where the CPU has spare cycles on average, or by *sustained overload* where the offered load over a long period of time is more than the CPU can handle. In either case, it may be possible to solve the problem by adjusting the mapping of tasks to execution environments using the strategies outlined in Figure 2.

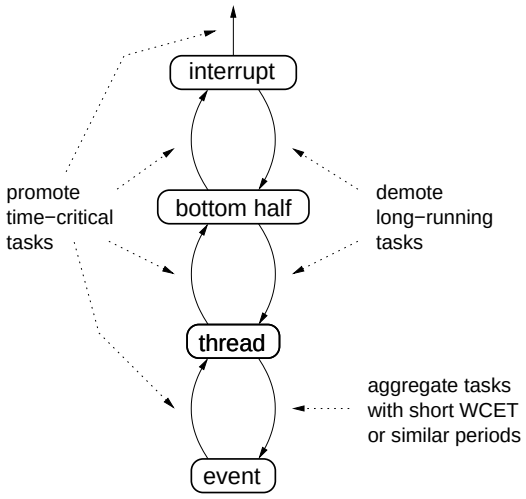

Figure 2. Movement of tasks between execution environments in an RTOS

Transient overload For a system that is suffering from transient overload, developers must identify a task that is missing deadlines and also the task or tasks that cause the delay. There are then three options: "promote" the code that is missing deadlines so that it runs in a higher priority execution environment, "demote" the code that is causing deadlines to be missed, or adjust priorities within a single execution environment. In Section 4 we give examples of applying both promotion and demotion in a real system. Promoting code that is already running in an interrupt environment can be accomplished by virtualizing the interrupt controller, a technique used in RTLinux [20].

Sustained overload When there is sustained overload, the CPU cannot keep up. This is a more difficult problem than transient overload because it cannot be solved solely by adjusting priorities: it can only be dealt with by reducing system overhead. Overhead can be reduced by moving a task to a place in the scheduling hierarchy where it will be dispatched by a scheduler that has less overhead than its current scheduler, or by moving a task to a place where it incurs less synchronization overhead. For example, processing incoming network packets in thread context requires frequent, expensive thread dispatches. This overhead can be reduced by running network code in a bottom-half handler, and in fact this is what real operating systems like Linux do. Another example concerns two tightly coupled tasks, each executing in its own thread. These might benefit from being co-located in the same thread in order to avoid the overhead of acquiring locks when accessing shared resources.

3. Reasoning about execution environments

There are two main challenges in building, analyzing, and evolving systems containing hierarchies of execution environments. In Section 3.1 we address real-time analysis of restricted scheduling hierarchies. Key issues are "flattening" the scheduling hierarchy into a form where it can be analyzed using traditional real-time analysis and deriving a blocking term for each task in the system. In Section 3.2 we address concurrency issues — how can we help developers create systems that are free of race conditions? The problem is difficult because there are often several choices of lock implementation, such as thread mutexes and disabling interrupts. For some critical sections only one choice of lock is correct. In other cases there are several valid choices, each with different performance characteristics. For example, disabling interrupts instead of taking a thread mutex

may increase overall throughput but also increase the number of missed deadlines.

3.1. Response time analysis for hierarchical priority schedulers

This section shows how to "flatten" a scheduling hierarchy into a form where it can be analyzed using a standard static priority analysis [2, 11], and informally argues that our algorithms are correct. Since the goal is to encompass real-time systems as they are typically built — that is, primarily using priorities — the analysis that we propose is simple when compared to more general frameworks for reasoning about hierarchical scheduling such as PShED [12] or HLS [15].

Algorithm 1: Preemptive priority schedulers This algorithm applies to hierarchies containing only preemptive static priority schedulers. It assigns priorities $0..n-1$ to a hierarchy with n tasks at the leaves, assuming that zero is the highest priority and that schedulers consume negligible CPU time. View the scheduling hierarchy as a directed acyclic graph and perform a depth-first traversal, visiting the children of each scheduler in order of highest to lowest priority. The algorithm maintains a counter that initially has value zero. Each time the traversal visits a leaf of the hierarchy, it assigns a priority to the task equal to the value of the counter, and then increments the counter.

To see that this algorithm is correct, consider what would happen if, initially, all tasks were runnable. The scheduler at the root of the hierarchy would select its highest priority child and so on until a leaf task was reached. Notice that this task is the first task that would be visited by our algorithm, and hence the task that is assigned priority 0. Next, consider what would happen if the highest-priority task completes or blocks: the hierarchy selects the next-highest priority task and runs it. This task corresponds to the second task visited by our algorithm. The argument can be repeated until all tasks have been visited.

Algorithm 2: Adding support for non-preemptive FIFO and priority schedulers Many operating systems, such as Windows 2000, Linux, and TinyOS, use a non-preemptive scheduler to run tasks that run for too long to be placed in interrupt context, but that still require lightweight dispatching. These schedulers typically use a FIFO discipline to avoid starvation, but may also support priorities (e.g., the Windows 2000 kernel supports a hybrid priority/FIFO discipline for scheduling deferred procedure calls). Our second algorithm supports analysis of scheduling hierarchies containing non-preemptive schedulers with the restriction that these schedulers only run leaf tasks, not other schedulers.

We make use of the response time analysis developed to analyze task sets scheduled with preemption thresholds [17]; it is useful because it permits analysis of a structured mix of preemptive and non-preemptive scheduling. For preemptive priority schedulers, proceed as for Algorithm 1, additionally assigning each task a preemption threshold equal to its priority. For each FIFO scheduler, all tasks are assigned the same priority, and preemption thresholds are set to the same value. For each non-preemptive priority scheduler, assign priorities as for preemptive schedulers, but set the preemption threshold for each task to the value of the highest priority of any task run by that scheduler.

Since this algorithm degenerates to Algorithm 1 when all schedulers are preemptive, we will only argue the correctness of handling non-preemptive schedulers. First, notice that, at worst, a task scheduled by a FIFO scheduler is queued up behind one instance of each other task run by that scheduler. This situation can be modeled by assigning the same priority to all tasks. However, it is necessary to make a small change to Saksena and Wang's response time analysis for task sets with preemption thresholds since they assumed that priorities are assigned uniquely. This can be accomplished in the same way that it has been accomplished in other response time analyses: by ensuring that the analysis accounts for interference from same-priority tasks in addition to interference from higher priority tasks. Second, notice that for non-preemptive priority schedulers, priorities are assigned in a continuous range and that their preemption threshold is set to the highest priority in that range. This gives us two key properties. First, within the set of tasks run by a non-preemptive priority scheduler no preemption is possible because when a task starts to run its priority is elevated to its preemption threshold, which is too high to permit any other task run by that scheduler to preempt it. Second, the presence of a non-preemptive scheduler does not affect preemption relations anywhere else in the system. To see this, notice that while the effect of elevated preemption thresholds is to suppress preemption relations, no task has a preemption threshold high enough to cause it to interfere with a task run by another scheduler.

The example scheduling hierarchy in Figure 3 shows the priority and preemption threshold that would be computed for each task using Algorithm 2.

Accounting for scheduling overhead In priority-based systems context switch overhead is accounted for by "charging" the cost of two context switches to each instance of a real-time task. This concept extends straightforwardly to hierarchies of priority schedulers. To compute the context switch cost, add up the costs of a context switch performed by each scheduler between the root of the scheduling hierarchy and the given task. So, for example, an in-

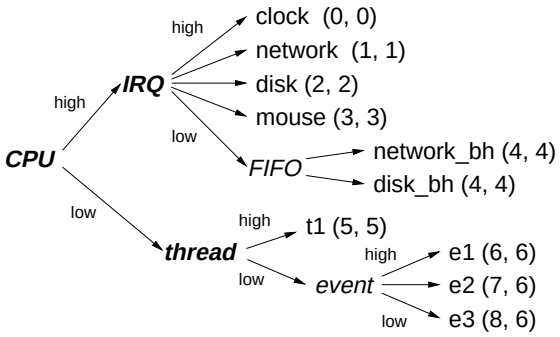

Figure 3. Priorities and preemption thresholds computed by Algorithm 2

terrupt handler is charged only for the cost of the interrupt prologue and epilogue, but event handler e1 in Figure 3 is charged for the cost of scheduling a thread and also dispatching an event.

Computing blocking terms Blocking terms [18] account for the worst-case duration of priority inversion in a real-time system, and they are significant in practice because different schedulers may have radically different blocking terms. For example, empirical evidence suggests that the Linux kernel can delay the execution of high-priority threads by up to 50 ms [13], while RTLinux [20] claims to have worst-case scheduling latency about three orders of magnitude less than this.

To compute the blocking term for a task in a system with hierarchical scheduling, observe that each scheduler between the task and the root of the scheduling hierarchy has an opportunity to block the task. Label each edge in the hierarchy with the blocking term contributed by its parent scheduler. The overall blocking term for any task is the sum of the blocking terms of all edges between it and the root of the hierarchy.

3.2. Concurrency analysis

This section describes TSL, a formal system we have created to address locking concerns in systems with multiple execution environments where it is possible to statically identify tasks, resources, and locks. Many embedded and real-time systems are static in this sense. TSL is based on two main ideas. First, a conservative estimation of the execution environment or environments that each function may execute in can be found by examining the call graph for a system. Second, the relationships between execution environments can be expressed in terms of hierarchical scheduling and asymmetrical preemption relations.

Tasks Tasks are sequential flows of control through a system. Some tasks finish by returning control to the scheduler that invoked them; other tasks encapsulate an infinite loop and these never finish — control only returns to their scheduler through preemption. Some tasks act as schedulers by passing control to tasks running below them in the hierarchy. Familiar examples of tasks are interrupt handlers, event handlers, and threads.

Schedulers and preemption Schedulers are responsible for sequencing the execution of tasks. TSL models schedulers in a modular way by specifying the preemption relations that the scheduler induces between tasks that it schedules. Preemption relations are represented asymmetrically: we write $t_1 \not\downarrow t_2$ when task t_2 may preempt task t_1 in some execution of the system. That is, if we cannot rule out the possibility that t_2 can start to run after t_1 begins to execute but before t_1 finishes. This simple definition suits our present needs because it is directly related to preemption, a main concern of TSL. In the future, to support multiprocessors and other hardware-based forms of concurrency, we will likely adopt a definition of atomicity based on sequential consistency.

The simplest scheduler, a non-preemptive event scheduler, does not permit any child to preempt any other child. For any two children t_1 and t_2 of such a scheduler, $\neg(t_1 \not\downarrow t_2) \wedge \neg(t_2 \not\downarrow t_1)$. On the other hand, a generic preemptive scheduler, such as a UNIX time-sharing scheduler, potentially permits each child task to preempt each other child task. That is, for any two children of such a scheduler, $t_1 \not\downarrow t_2 \wedge t_2 \not\downarrow t_1$. A third type of scheduler commonly found in systems software is a strict priority scheduler such as the interrupt controller in a typical PC. It schedules a number of tasks $t_1..t_n$ and it is the case that $t_j \not\downarrow t_i$ only when $i < j$. Most software-based priority schedulers are not strict — they permit priority inversion due to blocking.

Locks Preemption can lead to race conditions, and locks are used to eliminate problematic preemption relations. Tasks in TSL hold a (possibly empty) set of locks at each program point. We write $t_1 \not\downarrow_l t_2$ if parts of a task t_2 that hold a lock l can start to run while a task t_1 holds l ($t_1 \not\downarrow t_2$ should be read as shorthand for $t_1 \not\downarrow_\varnothing t_2$). For example, consider two threads that can usually preempt each other. If holding a thread lock lk blocks a task t_2 from entering critical sections in t_1 protected by lk, then $(t_1 \not\downarrow_\varnothing t_2) \wedge \neg(t_1 \not\downarrow_{lk} t_2)$.

Every lock is provided by some scheduler; the kinds of locks provided by a scheduler are part of its specification. We write $t \multimap l$ if a scheduler t provides a lock l, and require that each lock be provided by exactly one scheduler. There are two common kinds of locks. First, locks that resemble disabling interrupts: they prevent any task run by

a particular scheduler from preempting a task that holds the lock. Second, locks that resemble thread mutexes: they only prevent preemption by tasks the hold the same instance of the type of lock. Some tasks, such as interrupt handlers, implicitly hold locks when they start running. Accounting for these locks is a matter of specification and needs no special support in TSL.

Resources At each program point a task is accessing a (possibly empty) set of resources. We write $t \rightarrow_L r$ if a task t potentially uses a resource r while holding a set of locks L. Resources represent data structures or hardware devices that must be accessed atomically.

Races The definition of a race condition is as follows:

$$\begin{array}{rcl} \mathit{race}(t_1, t_2, r) & \stackrel{\text{def}}{=} & t_1 \rightarrow_{L_1} r \\ & \wedge & t_2 \rightarrow_{L_2} r \\ & \wedge & t_1 \neq t2 \\ & \wedge & t_1 \not\downarrow_{L_1 \cap L_2} t_2 \end{array}$$

That is, a race can occur if two different tasks t_1 and t_2 use a common resource r with some common set of locks $L_1 \cap L_2$, and if t_2 can preempt t_1 even when t_1 holds those locks. For example, if some task t_1 uses a resource r with locks $\{l_1, l_2, l_3\}$ and another task t_2 uses r with locks $\{l_2, l_3, l_4\}$ then they hold locks $\{l_2, l_3\}$ in common and a race occurs if and only if $t_1 \not\downarrow_{\{l_2, l_3\}} t_2$.

Hierarchical scheduling Each scheduler is itself a task from the point of view of a scheduler one level higher in the hierarchy. For example, when an OS schedules a thread, the thread is considered to be a task regardless of whether or not an event scheduler is implemented inside the thread. We write $t_1 \lhd t_2$ if a scheduler t_1 is directly above task t_2 in the hierarchy; $\lhd$ is the *parent* relation. Similarly, the *ancestor* relation $\lhd^+$ is the transitive closure of $\lhd$. TSL's definition of "task" is slightly more general than the sense usually used by the real-time community, which does not encompass scheduling code.

TSL gains much of its power by exploiting the properties of hierarchies of schedulers. First, the ability or lack of ability to preempt is inherited down the scheduling hierarchy: if a task t_1 cannot preempt a task t_2, then t_1 cannot preempt any descendent of t_2. A useful consequence is that if the *nearest common scheduler* in the hierarchy to two tasks is a non-preemptive scheduler, then neither task can preempt the other.

When a task that is the descendent of a particular scheduler requests a lock, the scheduler may have to block the task. When this happens, the scheduler does not directly block the task that requested the lock, but instead blocks its currently running child, which must be transitively scheduling the task that requested the lock. If a task attempts to acquire a lock that is not provided by one of its ancestors in the scheduling hierarchy then there is no child task for the scheduler to block — an illegal action has occurred. Using TSL, we can check for this generalized version of the "blocking in interrupt" problem by ensuring that tasks only acquire blocking locks provided by their ancestor schedulers. We formalize this generalization as follows:

$$\begin{array}{rcl} \mathit{illegal}(t, l) \stackrel{\text{def}}{=} \exists\ t_1. & & t_1 \multimap l \\ & \wedge & \neg(t_1 \lhd^+ t) \\ & \wedge & t \rightarrow_L r \\ & \wedge & l \in L \\ & \wedge & \mathit{blocking}(l) \end{array}$$

Execution environments An execution environment is a context for running application code that is created by an instance of a scheduler. Our operating definition of *execution environment* is informal: a list of attributes of the environment that are of practical significance to software developers. The properties of an environment are determined primarily by its scheduler, but also by the rest of the system and even the underlying hardware platform. For example, consider an execution environment created by a non-preemptive event scheduler that runs in interrupt context:

- The non-preemptive parent scheduler implies that tasks in the environment cannot preempt each other.
- The lack of an ancestor scheduler that supports blocking implies that tasks cannot block.
- The details of the compiler and underlying hardware platform determine the number of microseconds of overhead in taking an interrupt and dispatching an event.
- The longest WCET of any task in the environment, added to the longest duration that any task in the system disables interrupts for, determines the worst-case scheduling latency for the highest-priority event in the environment.

Reasoning about systems In this paper we assume that the code used to build a system is annotated with TSL properties such as accesses to resources and specifications for schedulers. We currently do this mostly by hand, but in the future we plan to develop better tool support for deriving facts about code. To verify that a system does not contain any race conditions or illegal blocking we run the *TSL checker*, a lightweight automatic theorem prover. The domain of TSL facts is finite, so simple algorithms can be used; our current TSL checker is a forward-chaining evaluator written in Haskell.

One of our chief goals with TSL is to enable modular, reusable specifications of schedulers, tasks, and applications. TSL as it currently exists comes reasonably close to this goal but there are two places where we have experienced problems. First, on adding a new task to a scheduler, one must add any preemption relations between the task and its sibling tasks. In raw TSL this would require that we modify the scheduler specifications as the application changes, or the application specifications as the scheduler hierarchy changes — neither approach is very modular. To avoid this, the TSL checker provides direct support for common types of schedulers, automatically generating the necessary preemption relations. Second, TSL does not account for the ability of tasks to release other tasks. For example, although schedulers implement time-slicing using timer interrupts, TSL misses the fact that disabling interrupts suppresses time slicing. At present one needs to add non-modular details about task releases to TSL models to keep it from returning too many false positives — potential race conditions that cannot actually happen. We are working on adding support for causality in task releases to make it possible to infer these properties from first principles.

In previous work [14] we focused on *synchronization inference*: the automatic derivation of an appropriate lock implementation for each critical section in a system. However, in this paper we focus on using TSL only to look for problems in a system, relying on the developer to fix these problems. TSL reports the presence of illegal locking or race conditions; when doing so it provides a list of problematic lock acquisitions or preemption relations.

4. Evolving a real system

This section demonstrates and validates our approach to creating evolvable real-time systems by solving problems meeting real-time deadlines in TinyOS [9], a simple component-based operating system for networked sensor nodes. TinyOS runs on "motes," small microcontroller-based systems equipped with a wireless packet radio. We consider two changes to the base system:

Adding long running tasks: We show how to support long-running sections of code without disrupting ongoing computations.

Meeting radio deadlines: We solve a design problem in TinyOS where critical sections cause blocking that interferes with reliable radio reception.

4.1. TinyOS and the ping/pong application

The TinyOS scheduling hierarchy, shown in Figure 4, is analogous to the bottom half of an RTOS or general-purpose operating system. It runs in a single address space and contains two execution environments: interrupt handlers running at high priority and tasks that are scheduled in FIFO order at low priority. Tasks provide a level of deferred processing, reserving interrupt context for short, time-critical sections of code. In the context of TinyOS, *task* has a more restricted meaning than its usual definition: it does not refer to interrupt handlers or scheduling code.

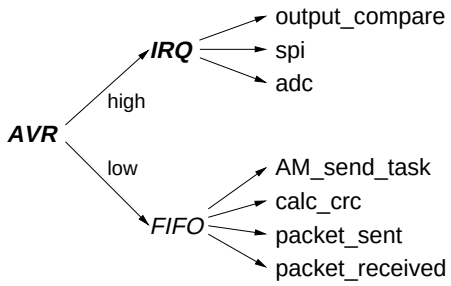

Figure 4. The TinyOS scheduling hierarchy: interrupts are scheduled preemptively by the CPU while tasks are scheduled in software and have run-to-completion semantics

Our motivating application is simple, but exercises the entire functionality of the TinyOS network stack. The application runs on a pair of motes: the first sends out packets and counts replies; the second returns any packet it receives to the first. The performance metric of interest is overall throughput: an important factor for networked sensors.

Modeling the application in TSL Since embedded systems typically have a relatively small number of schedulers and tasks, it is straightforward to develop a model of the scheduling aspects of an application. In fact, the figures we have used to illustrate scheduling hierarchies contain almost all of the required information. The main remaining challenge in developing a TSL model is tracking the use of resources and locks inside the code. One approach is to annotate every critical section with which locks it uses and which resources it accesses, and then to use a tool that we have developed to perform a simple source-code analysis to extract the callgraph of the application. This is quite feasible; for example, the authors of TinyOS have developed a new C-like language nesC [7] in which critical sections are explicitly identified; it would take little more effort from the programmer to identify which resources are used by each critical section. This approach works best if it is applied as the code is written and is maintained with the code. Since we did not write most of our application code and wish to be able to track future changes with little effort, we judged that manually annotating the code in this fine-grained way was not desirable.

As an engineering compromise, we developed a coarse-grained model of the system based entirely on the scheduling hierarchy, with the goal of detecting race conditions introduced by evolving the system, but without the ability to detect race conditions that are in the original system. Our model consists of a single resource used by all interrupt handlers and all TinyOS tasks when they have interrupts disabled, and a single resource used by all TinyOS tasks that have interrupts enabled. This is a very coarse-grained model so it is worth pointing out that coarse-grained models are actually more sensitive to race conditions than are fine-grained models, since any approximations made must be safe ones. As one would expect, when using this simple model TSL reported no race conditions in the base application.

Experimental setup All experiments were run on TinyOS version 0.6. We used Mica motes based on the Atmel ATmega103 8-bit microcontroller running at 4 MHz with 4 KB of RAM and 128 KB of flash memory. These motes have a radio theoretically capable of sending 115 kbps; TinyOS drives it at 40 kbps. Timing measurements were taken by setting output pins on the AVR and observing the results using a logic analyzer.

4.2. Supporting long-running tasks

In this experiment, we investigate how we can add long-running tasks while maintaining high radio throughput. This is a desirable feature as mote hardware becomes capable of running sophisticated algorithms such as those supporting signal processing and cryptography.

The experiment We used a timer interrupt to post a long-running task four times per second. The task is synthetic: it makes a copy of the packet most recently received over the radio and then spins for the rest of its execution time. The experimental procedure was to vary the run-time of the long-running task while measuring network throughput.

The "original TinyOS" data points in Figure 5 show how the long-running task interferes with throughput. We do not have data points for 10 and 25 ms because the TinyOS network subsystem consistently crashes in those experiments due to a bug, or at 200 ms because no packets were reliably returned at (or beyond) this point.

Our hypothesis was that the drop in throughput was primarily caused by a delay in returning the packet to the sender. We believed that most packet transmission and receipt processing runs in interrupt context. Since code run in interrupt context is not affected by our long-running task, the data clearly indicate that some part of packet processing runs as a task. By examining the code, we quickly confirmed that the pong application posts tasks as part of sending and receiving packets. Since TinyOS schedules tasks non-preemptively, transmission of the packet will be delayed until the load task completes if the load task is running when the packet arrives.

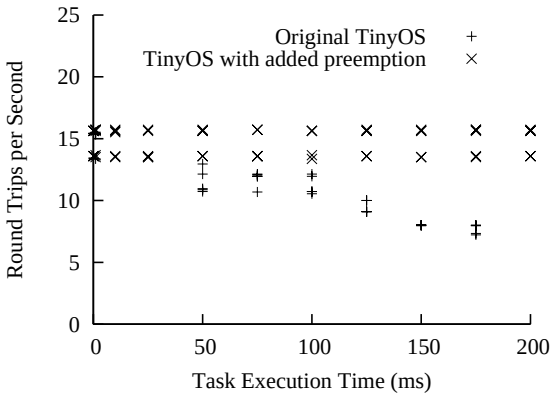

Figure 5. Impact of posting a long-running task four times per second

Restructuring the scheduling hierarchy We now illustrate how to apply our method for restructuring a system; it was described in Section 2.4. The goal is to prevent TinyOS radio tasks from missing their deadlines. To see where the problems begin, we performed response time analysis for the four TinyOS radio tasks and a variable-length CPU-intensive task that are run by TinyOS's non-preemptive FIFO scheduler. This analysis showed that if the long-running task runs for more than 3.68 ms it cannot be run in a standard version of TinyOS without causing missed radio deadlines. To avoid missed deadlines, the long-running task must be demoted or the radio tasks must be promoted.

We decided not to promote radio code into interrupt context because the TinyOS task that computes the CRC field for an outgoing packet runs for too long: more than 3 ms. Rather, we demoted the long-running task, necessitating significant changes to the structure of execution environments in TinyOS. To make these changes, we multi-instantiated the TinyOS task scheduler and then ran each scheduler instance in a separate preemptive thread provided by the AvrX [3] operating system. Unlike TinyOS, which does not support threads, AvrX is a typical RTOS: it provides preemptive multithreading, mutual exclusion, timers, message queues, etc.

The resulting scheduling hierarchy is depicted in Figure 6. It contains two task schedulers: a foreground scheduler that runs high-priority network tasks and a background scheduler that runs long-running tasks (of which there is only one in our experiment). Tasks running on each sched-

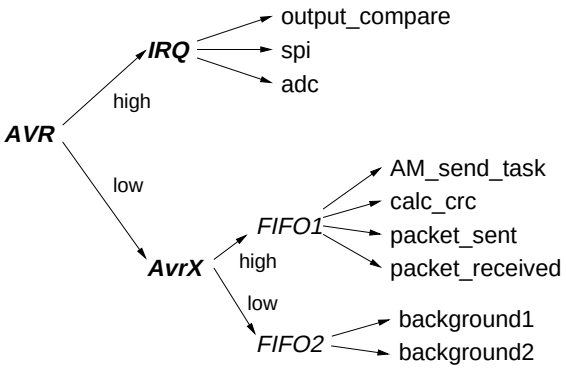

Figure 6. TinyOS scheduling hierarchy after demoting long-running tasks

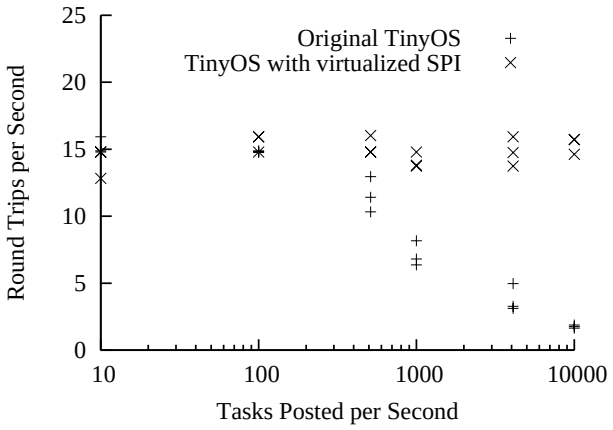

Figure 7. Posting tasks interferes with radio reception in the standard TinyOS kernel

uler retain run-to-completion semantics with respect to each other, but a foreground task can preempt a background task.

Validating the changed system There are two components to validating the system after making a change to its execution environments: checking that the real-time problem that motivated the change was fixed, and ensuring that race conditions were not added to the system. We hypothesized that network throughput would be restored to its original level if we could avoid delaying the execution of TinyOS tasks that support packet processing, and this hypothesis is confirmed in Figure 5.

To check for race conditions we modeled the new scheduling hierarchy and the long-running task in TSL. Since we wrote the long-running task ourselves, it seemed reasonable to use a finer-grained model of resources and so we added an additional resource representing the packet buffer and noted that it was accessed by both the long-running task and a task in the TinyOS network stack. We then ran the TSL checker over the modified system; it told us that there was, in fact, a race condition caused by the new preemption relation between the radio tasks and the long-running task. After fixing this race by protecting the access with a mutex and updating the model, TSL was not able to find any more errors.

4.3. Meeting radio deadlines

The original mica radio stack [9] transferred data from the radio hardware one bit at a time. Recent TinyOS versions use the serial peripheral interface (SPI) hardware to implement byte-level transfers from the radio to the host CPU, increasing link speed and reducing CPU overhead. However, interrupts signaled by the SPI have an extremely short deadline: if the SPI interrupt is not handled within 22 CPU cycles (5.5 μs) of its arrival, a byte of radio data is lost. Furthermore, losing a byte is almost certain to force an entire packet to be lost because it defeats the SEC/DED payload-encoding scheme that can tolerate a single corrupted bit per byte. A consequence of this extremely tight real-time deadline is that posting a TinyOS task, which disables interrupts for a few cycles in order to be safely callable from both interrupt and non-interrupt environments, can cause the SPI deadline to be missed. Experiments show that posting a task a few hundred times per second causes a significant drop in radio performance, as demonstrated in Figure 7.

Changing the hierarchy To maintain radio throughput while posting tasks we promote part of the SPI interrupt handler by performing a lightweight virtualization of the interrupt handling structure of TinyOS in a manner analogous to what was done in RTLinux [20]. The virtualization has two aspects. First, by default TinyOS enforces mutual exclusion by disabling interrupts, or in other words, taking a lock provided by the scheduler at the root of the hierarchy. As we saw in Section 3.1, this contributes a blocking term to every task in the system. To avoid applying a large blocking term to the SPI interrupt handler, mutual exclusion in TinyOS must be implemented using a lock provided by a scheduler one level lower in the scheduling hierarchy — the virtual interrupt scheduler. This change is shown in Figure 8; it can be effected by modifying several macros in the TinyOS header files.

Second, the SPI interrupt handler must be split into two parts: one that reads a byte from the SPI register and another that integrates this byte into the ongoing computation. The first part is time critical and must proceed without delay no matter what the rest of the system is doing. The second part interacts with the rest of the system and there-

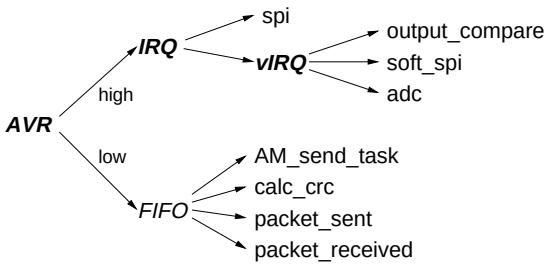

Figure 8. Virtualizing the interrupt scheduler exempts the SPI interrupt from being disabled by critical sections, and hence it is able to reliably meet real-time deadlines

fore it must be synchronous with respect to TinyOS as a whole. It is not time critical. The implementation that we chose is for the time critical part of the SPI interrupt handler to trigger a software interrupt. If an SPI interrupt arrives while TinyOS is not in a critical section, then the software interrupt handler runs immediately after the SPI interrupt handler returns. However, if an SPI interrupt arrives while TinyOS is in a critical section, then the execution of the soft interrupt will be delayed until the critical section is finished. The important property of this hierarchy is that interrupts run by the vIRQ scheduler are disabled by critical sections in TinyOS code, while interrupts run by the IRQ scheduler are not.

Validating the changed system Again, to validate the changed system we must show that the real-time problem has been solved and that no new concurrency errors have been added. Since posting a task in our modified TinyOS kernel cannot cause missed SPI interrupts, radio throughput should not vary as a function of the frequency at which tasks are posted. The data points in Figure 7 that correspond to the modified kernel show that this is the case.

By introducing many new preemption relations we have made a potentially serious change to the scheduling hierarchy. Indeed, TSL detected a race condition in the modified system. The problem is that the new software interrupt handler can be preempted by the SPI interrupt handler and they share a small amount of state. Since these two tasks are causally related — the software interrupt is triggered by the hardware interrupt — we reasoned that this was a false alarm since the race condition can only occur if the software interrupt is delayed for 200 μs: long enough for another byte to arrive. Since the packet will be lost anyway in this case, we opted not to eliminate the race condition by protecting accesses to the shared variable — this would add overhead without providing any benefit.

5. Related work

A number of research projects have used hierarchical scheduling techniques to create flexible real-time systems. These include PShED [12], the open environment for real-time applications in Windows NT [4], hierarchical fixed-priority scheduling [16], hierarchical virtual real-time resources [5], and our own HLS [15]. These previous projects proposed new hybrid scheduling algorithms that provide new and useful reservation-like guarantees to real-time applications and collections of applications in open systems where task characteristics are not always known in advance. Our current work, on the other hand, is fundamentally different: its goal is to describe and analyze the hierarchical priority schedulers that are already present in essentially all real-time and embedded systems. It focuses on closed, static systems where task characteristics are known in advance, and it supports concurrency analysis.

The trend towards inclusion of concurrency in mainstream language definitions such as Java and towards strong static checking for errors is leading programming language research in the direction of providing annotations [8] or extending type systems to model locking protocols [6]. These efforts, however, have not addressed the possibility of multiple execution environments as we have in TSL.

Several RTOSs such as RTLinux [20] and TimeSys Linux/GPL [19] have restructured scheduling hierarchies, or altered the mapping of tasks to schedulers, in order to help software meet real-time deadlines. While these efforts have focused on implementing or evaluating a single transformation, we have developed a general framework and accompanying analyses for restructuring systems code to fix problems in meeting real-time deadlines. We have also made the execution environments created by the hierarchical schedulers in a system into a first-class concept, which had not been done before.

Some aspects of multiple execution environments have been addressed by previous research. For example, it has long been known that static priority analysis permits homogeneous analysis of the ability of interrupts and threads to meet real-time deadlines. Jeffay and Stone analyzed the schedulability of a mix of interrupt handlers and threads scheduled using EDF [10]. Saksena and Wang showed how to use preemption threshold scheduling to map abstract tasks onto both preemptive threads and non-preemptive user-level events [17], allocating a minimal number of preemptive threads. However, these efforts have focused on a limited set of execution environments, and none of them have addressed the concurrency issues created by a diversity of execution environments, as we have with TSL.

6. Conclusion

We have made several contributions towards the creation of evolvable real-time systems. First, we have shown that it is useful to look at a system as a collection of execution environments with different properties that can be exploited by developers. We have also provided heuristics for mapping tasks to execution environments in such a way that real-time deadlines are likely to be met. Our second contribution is two novel algorithms that make it possible to perform real-time analysis of hierarchies of schedulers containing priority and FIFO schedulers. Third, we have shown that a whole-program concurrency analysis based on task scheduler logic can be used to detect race conditions and other concurrency errors resulting from changing the mapping of tasks to environments. Finally, we have validated our approach to evolving real-time systems using two case studies that evolve the structure of networked sensor node software.

Acknowledgments: The authors would like to thank Eric Eide, Mike Hibler, and the reviewers for their helpful comments on this paper.

This work was supported, in part, by the National Science Foundation under award CCR-0209185 and by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreements F30602-99-1-0503 and F33615-00-C-1696.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP)*, pages 95–109, Pacific Grove, CA, Oct. 1991.

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

[3] L. Barello. The AvrX real time kernel. `http://barello.net/avrx`.

[4] Z. Deng, J. W.-S. Liu, L. Zhang, S. Mouna, and A. Frei. An open environment for real-time applications. *Real-Time Systems Journal*, 16(2/3):165–185, May 1999.

[5] X. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proc. of the 23rd IEEE Real-Time Systems Symp. (RTSS)*, Austin, TX, Dec. 2002.

[6] C. Flanagan and M. Abadi. Types for safe locking. In S. Swierstra, editor, *ESOP'99 Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, Mar. 1999.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[8] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *Proc. of the 24th Intl. Conf. on Software Engineering (ICSE)*, pages 453–463, Orlando, FL, May 2002.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, Nov. 2000.

[10] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proc. of the 14th IEEE Real-Time Systems Symp. (RTSS)*, pages 212–221, Raleigh-Durham, NC, Dec. 1993.

[11] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[12] G. Lipari, J. Carpenter, and S. K. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard real-time environments. In *Proc. of the 21st IEEE Real-Time Systems Symp. (RTSS)*, pages 217–226, Orlando, FL, Nov. 2000.

[13] J. Regehr. Inferring scheduling behavior with Hourglass. In *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pages 143–156, Monterey, CA, June 2002.

[14] J. Regehr and A. Reid. Lock inference for systems software. In *Proc. of the Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Boston, MA, Mar. 2003.

[15] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. of the 22nd IEEE Real-Time Systems Symp. (RTSS)*, pages 3–14, London, UK, Dec. 2001.

[16] S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proc. of the 14th IEEE Euromicro Conf. on Real-Time Systems*, Vienna, Austria, June 2002.

[17] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symp. (RTSS)*, Orlando, FL, Nov. 2000.

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

[19] TimeSys Linux/GPL. `http://timesys.com`.

[20] V. Yodaiken. The RTLinux manifesto. In *Proc. of The 5th Linux Expo*, Raleigh, NC, Mar. 1999.