Preventing Interrupt Overload

John Regehr Usit Duongsaa
School of Computing, University of Utah
{regehr,duongsaa}@cs.utah.edu

Abstract

Performance guarantees can be given to tasks in an embedded system by ensuring that access to each shared resource is mediated by an appropriate scheduler. However, almost all previous work on CPU scheduling has focused on thread-level scheduling, resulting in systems that are vulnerable to a lower-level form of overload that occurs when too many interrupts arrive. This paper describes three new techniques, two software-based and one hardware-based, for creating systems that delay or drop excessive interrupt requests before they can overload a processor. Our interrupt schedulers bound both the amount of work performed in interrupt context and its granularity, making it possible to provide strong progress guarantees to thread-level processing. We show that our solutions work and are efficient when implemented on embedded processors. We have also taken a description for a microprocessor in VHDL, modified it to include logic that prevents interrupt overload, synthesized the processor, and verified that it works using simulation. By allowing developers to avoid making assumptions about the worst-case interrupt rates of peripherals, our work fills an important gap in the chain of reasoning leading to a validated system. These techniques cannot replace careful system design, but they do provide a last-ditch safety guarantee in the presence of a serious malfunction.

Categories and Subject Descriptors C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems; D.4 [*OPERATING SYSTEMS*]: Process management; B.8 [*PERFORMANCE AND RELIABILITY*]: Performance analysis and design aids

General Terms Performance, design, reliability

Keywords Interrupts, overload, scheduling, embedded

1. Introduction

Many interrupt-driven embedded systems are vulnerable to *interrupt overload*: the condition where external interrupts are signaled frequently enough that other activities running on a processor are starved. Interrupts are dangerous because they are implicitly given higher priority than processing done in other contexts such as threads. Also, as we show in Section 2, some common interrupt sources have very high maximum arrival rates. Neglecting to bound maximum interrupt arrival rates creates an important gap in the chain of assumptions leading to a high-assurance system.

This paper presents a collection of techniques that can lead to systems that actively prevent interrupt overload. Our specific goals were to develop interrupt schedulers that have the following properties:

- Average processor load must be proportional to the interrupt arrival rate during underload.
- Average processor load must be bounded by a constant during overload.
- It must be possible to quantify the worst-case delays incurred by low-priority interrupts and non-interrupt work, in order to give them performance guarantees.
- The schedulers should be dynamically parameterizable, in order to permit systems to flexibly choose appropriate levels of protection from interrupt overload.
- Overhead must be reasonable.

Interrupt schedulers meeting these goals could be implemented in software or hardware; we explore both strategies. All of our schedulers should work smoothly in concert with other schedulers such as network bandwidth reservations or thread-level CPU reservations.

Our approach to eliminating interrupt overload is motivated by three main characteristics of embedded systems. First, embedded systems are often highly resource-limited. In fact, due to memory constraints, many small embedded systems such as energy-efficient sensor network nodes based on 8-bit microcontrollers lack high-level schedulable contexts like threads and processes. This limits the utility of the many reservation-based abstractions that have been developed to prevent CPU overload [12, 15, 18] in open systems. Also, thread-level reservations cannot directly prevent interrupt overload, though they can delay its onset if as much interrupt-mode code as possible is migrated into threads. However, adding thread dispatches to the critical path for interrupt handling is itself a problem on slow processors where thread dispatch is relatively expensive. For example, as part of a previous research effort [23] we attempted to run some TinyOS [10] radio interrupt code in thread mode. We found that this unacceptably delayed time-critical radio processing and, in addition, the next interrupt request would arrive before the previous thread invocation completed, leading to persistent CPU overload. Our solutions to interrupt overload directly throttle interrupt arrivals; they do not require code to be moved into different execution contexts.

Second, embedded systems tend to have strict timing requirements due to tight coupling between software and hardware. For example, sensor/actuator feedback loops can become unstable if software execution is delayed for too long, leading to system failure. Also, inexpensive embedded peripherals such as serial ports often impose real-time requirements on a system due to limited buffer size. Existing techniques for preventing overload in interrupt-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'05 June 15–17, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-018-3/05/0006...$5.00.

driven network subsystems [5, 19] drop excess packets as early as possible and switch from interrupt-driven to polling mode during overload. Dropping packets early—before copying them into a buffer, for example—can delay the onset of interrupt overload but cannot prevent it. Adaptively switching between interrupt-driven and polling I/O is not a generally suitable strategy for embedded systems with tight response time requirements: it does not offer any guarantees about progress between the time when overload begins and the time when the system responds. Our solutions focus on predictability and responsiveness: they permit strong performance guarantees to be made to non-interrupt work.

The third characteristic of embedded systems that motivates our work is their extreme cost sensitivity, which often leads them to use cheap peripherals with little or no onboard processing power. This implies that for most embedded systems, developers will not be able to modify device firmware to ensure that the main processor is not overloaded by interrupts. Our solutions to interrupt overload—in both software and hardware—all run on the main processor. In contrast, Druschel and Banga [5] and Dannowski and Härtig [4] have prevented interrupt overload by changing the firmware running on high-end NICs.

In the next section we describe potential causes for interrupt overload. We present our solutions to interrupt overload in Section 3, discuss additional interrupt scheduling issues in Section 4, and analyze the schedulers' behavior in Section 5. Section 6 presents the results of an experimental validation and evaluation of our schedulers and Section 7 describes a case study in protecting an embedded Ethernet device against network interrupt overload. Section 8 describes an experiment in synthesizing a hardware-based interrupt scheduler. We compare our work to related research in Section 9 and conclude in Section 10.

2. Interrupt Overload

The first moon landing was nearly aborted when a flood of radar data overloaded a CPU on the Lunar Landing Module, resulting in guidance computer resets [20, pp. 345–355]. The problem on Apollo 11 appears to have been caused by spurious signals coming from a disconnected device. It would not have been severe had the system been designed so that a single erroneous interrupt source was not permitted to overload the computer.

Embedded systems tend to be particularly interrupt-driven. One reason is that embedded systems are usually very cost-sensitive. This leads to the use of cheap, dumb peripherals that require constant micromanagement, with an extreme case being the canonical "bit-banged" network interface where each bit is sent over the wire using explicit software control. A second reason that interrupts are used heavily is that many processors are capable of going to sleep, greatly reducing power consumption, until an interrupt arrives. This is an important energy optimization for devices that rely on batteries. The obvious alternative to interrupts, polling, performs well during overload but degrades performance and consumes power during underload by generating useless work.

Interrupt overload is not necessarily caused by high interrupt loads, but rather by unexpectedly high interrupt loads. For example, a fast processor running software that performs minimal work in interrupt mode can easily handle hundreds of thousands of interrupts per second. On the other hand, a slow processor running lengthy interrupt code can be overwhelmed by merely hundreds of interrupts per second.

Computing a reliable maximum request rate for an interrupt source in an embedded system is difficult, often requiring reasoning about complex physical systems. For example, consider an optical shaft encoder used to measure wheel speed on a robot. The maximum interrupt rate of the encoder depends on the maximum speed of the robot and the design of the encoder wheel. However, what happens if the robot exceeds its maximum design speed, for example while going downhill? What if the encoder wheel gets dirty, causing it to deliver pulses too often?

Source	*Max. Interrupt Freq. (Hz)*
knife switch bounce	333
loose wire	500
toggle switch bounce	1 000
rocker switch bounce	1 300
serial port @115 kbps	11 500
10 Mbps Ethernet	14 880
CAN bus	15 000
I2C bus	50 000
USB	90 000
100 Mbps Ethernet	148 800
Gigabit Ethernet	1 488 000

Table 1. Potential sources of excessive interrupts for embedded processors. The top part of the table reflects the results of experiments and the bottom part presents numbers that we computed or found in the literature.

The data in Table 1 show some measured and computed worst-case interrupt rates. Even innocuous-seeming hardware, such as switches, can display interesting electrical behavior. For example, during the transition from open to closed and closed to open, switches that we measured (using a logic analyzer) created transient signals that an embedded processor would interpret as interrupt requests exceeding 1 kHz. This could easily cause problems for a system designed to handle only tens of switch transitions per second. The traditional way to debounce a switch is to implement a low-pass filter either in hardware or software. Although debouncing techniques are well-known to embedded systems designers, it is not enough just to debounce all switches: new and unforeseen "switches" can appear at run-time as a result of loose contacts or damaged wires. Both of these problems are more likely in embedded systems that operate in difficult environmental conditions with heat and vibration, and without routine maintenance. These conditions are, of course, very common—for example, in automobiles.

Network interfaces represent another potential source for interrupt overload. For example, consider an embedded CPU that exchanges data with other processors over 10 Mbps Ethernet using a specialized protocol that specifies 1000-byte packets. If the network interface interrupts on packet arrival, the maximum interrupt rate is 1.25 kHz. However, if a malfunctioning or malicious node sends minimum-sized (72 byte) packets, the interrupt rate increases to nearly 15 kHz [13], potentially starving important processing.

3. Preventing Interrupt Overload

This section briefly reviews interrupt handling and then presents our techniques for preventing interrupt overload. Two of these schemes operate entirely in software, and can be run on off-the-shelf microprocessors. The third technique is implemented in hardware.

3.1 Interrupt background

There is variation in the details of interrupt implementations: we describe the behavior of the Atmel AVR family of microcontrollers as it is typical and these are the processors that we use to evaluate our work in Section 6. Each interrupt has two special hardware bits associated with it: an enable bit and a pending bit. Also, there is a global interrupt enable bit that can be used to disable all interrupt handlers.

The CPU asynchronously polls the status of each interrupt request line. For an interrupt line whose firing condition is met—interrupts may be edge-triggered or level-triggered—the interrupt's pending bit is set. Then, before executing each instruction, the CPU checks the status of each interrupt's pending bit. If the global interrupt enable bit is cleared, the processor continues executing its normal instruction stream. If this bit is set, the lowest-numbered pending interrupt whose enable bit is set is selected for execution. The processor then atomically:

- clears the interrupt's pending bit,
- clears the global interrupt enable bit,
- pushes the program counter and processor status word onto the stack, and
- loads the address of the selected interrupt's vector into the program counter.

The CPU is now executing in interrupt mode and will continue to do so until a return-from-interrupt instruction is executed.

Interrupt requests may be lost in two ways. First, if an interrupt is triggered (that is, its firing condition is met) when its pending bit is already set, then the new interrupt request is lost. Second, some interrupt sources have no pending bit: if their triggering condition becomes false before the interrupt is handled, the interrupt request is lost.

Preventing interrupt overload amounts to stopping the processor from handling interrupts when developer-specified conditions are met. Although discarding some interrupts may result in poorer quality of service for the subsystem driven by that interrupt, we claim that this is the best option in the presence of erroneously high arrival rate. In other words, if it is advantageous to overall application performance to handle a higher rate of interrupts than the rate limit, then the limit should simply be increased.

A hardware-based implementation can simply filter undesirable signals out of the wire. Scheduling interrupts in software, on the other hand, must reduce to twiddling an interrupt's enable bit, as this is the only available scheduling mechanism. So, the basic difference between the hardware and software schedulers is that the former is logically outside the processor's interrupt arbitration logic while the latter is logically inside it.

3.2 Strict software scheduler

Our first technique for scheduling interrupt arrivals is implemented in software and is *strict*: it enforces a minimum interarrival time between interrupts. The minimum interarrival time or its inverse, the maximum interrupt frequency, is specified by system designers.

The algorithm is simple: the interrupt prologue is modified to include code clearing the interrupt's enable bit and setting a one-shot timer to expire one interarrival time in the future. When the timer expires, its handler re-enables the interrupt. Conflicting access to the timer (i.e., setting it when it is already set) is impossible. This solution works on unmodified hardware but incurs some overhead, doubling the number of interrupts handled.

3.3 Bursty software scheduler

The second software-based interrupt scheduler that we developed has lower overhead than the strict scheduler but provides weaker isolation. In effect, it is lazier, disabling an interrupt only after a burst of interrupt requests has been observed. This scheduler has two inputs: a maximum burst size and a maximum arrival rate for bursts of interrupts (as opposed to a maximum arrival rate for individual interrupt requests, as in the previous scheduler).

To implement this scheduler, the interrupt prologue is modified to increment a counter and, if the counter is greater than or equal to the burst size, clear the interrupt's enable bit because there is danger of overload. The interrupt counter is cleared by a periodic timer that runs asynchronously with respect to the interrupt workload; the frequency of this timer is the burst arrival rate. This timer also sets the interrupt enable bit if it was previously cleared.

A useful performance optimization is to leave the periodic timer interrupt disabled as long as the counter is below the threshold, avoiding timer overhead in the expected case where the threshold is seldom exceeded. This only works when a dedicated hardware timer is available for the interrupt scheduler; during underload its effect is to leave the timer interrupt pending almost all of the time. It is also necessary to clear the timer interrupt's pending bit just before enabling the timer interrupt, in order to avoid a bad case where three back-to-back bursts of device interrupts could otherwise occur.

The maximum burst size and the burst arrival rate can be tuned to produce different performance tradeoffs. For example, it is possible to maintain a constant asymptotic maximum interrupt arrival rate by increasing burst size and reducing burst arrival rate. This reduces overhead but, by permitting longer bursts, increases the amount of time that other code running in the system may be delayed.

An advantage of the bursty scheduler is that it permits the cost of timer interrupts to be amortized over a number of device interrupts, reducing overhead. A second, distinct, benefit is that some devices, such as network interfaces, are inherently bursty. It may be desirable to attempt to handle an entire burst, rather than handling only the first interrupt in a burst, or the first few, and dropping the rest, as the strict scheduler would do.

3.4 Hardware scheduler

Our final scheduling technique filters interrupt signals out of an interrupt request line before they ever reach the CPU's interrupt controller. In a sense this was the simplest of the three interrupt schedulers to design: since it runs in parallel with the main CPU, efficiency is not a major concern. We have two prototype implementations of the hardware scheduler: one on a second microcontroller, the other as a modified version of an embedded processor that is implemented as an FPGA (field programmable gate array) configuration. The algorithm described in this section applies to both prototypes.

The hardware scheduler uses a counter that is automatically decremented at a fixed rate. When an interrupt arrives, there are two possibilities:

1. The counter is at zero. In this case the counter is reset to an initial value and the interrupt is propagated to the CPU.
2. The counter is not at zero. In this case the interrupt arrival is merely noted. When the counter reaches zero, we are back to case one. There is no additional queuing: if several interrupts arrive while the counter is counting down, only one interrupt is delivered to the CPU when the counter reaches zero. Both the countdown frequency and the counter's initial value can be changed by the software.

The software overhead of this solution is zero: enforcement of minimum interarrival times is completely free. We will show in Section 8 that implementing a scheduler in hardware is cheap in terms of chip area.

3.5 Scheduling multiple interrupt sources

Scheduling multiple interrupt sources using a strict scheduler is as simple as replicating the hardware or software for each interrupt line. The bursty scheduler, on the other hand, presents interesting opportunities for optimization by using a single periodic timer to clear the counters for multiple interrupt sources.

Earlier we noted that for a bursty scheduler, it is important to choose a burst size that strikes the right balance between overhead

and protection from overload. There are more choices to make in a system with multiple interrupt sources. Simply picking the least common multiple (LCM) of the burst arrival rates for all interrupt sources is likely to result in a very high frequency and thus poor overall performance.

We can round up the maximum allowed rates to some multiples of a large number, allowing a single slow timer to service them all and still maintain reasonable protection. For example, a system with three sources with maximum arrival rates of 324, 200, and 754 Hz can be serviced by a single timer of 110 Hz and have a maximum burst size of 3, 2, and 7, respectively.

Also, note that the hardware timers provided by an embedded processor natively support only certain frequencies, so it is likely that some rounding will be required anyway.

4. Discussion

This section discusses a few additional issues in preventing interrupt overload.

Design space issues Although it would not be difficult to implement a bursty scheduler in hardware, we have not done so. The choices of strict vs. bursty and hardware vs. software are orthogonal: it is unlikely that we would have learned anything new by doing this. Similarly, we could have implemented a bursty scheduler that uses a one-shot timer, instead of a periodic timer, to reset the burst counter. Again, our judgment was that we wouldn't have learned anything new.

Interface issues A drawback of a software-based interrupt scheduler is that asynchronously modifying interrupt enable bits from timer callbacks is an inconvenience to developers who want to disable an individual interrupt source as a strategy for implementing mutual exclusion. In other words, we have made the interrupt enable bit into a shared variable, inviting race conditions. There are two solutions to this problem. The first is a hardware solution: each interrupt line could be outfitted with two enable bits, one that is set and cleared by the interrupt scheduling logic, the other being reserved for programmers. The interrupt would be permitted to fire only when both bits (and also the global interrupt enable bit) are set. The second solution is a software-based implementation equivalent to having two interrupt enable bits: the interrupt bit must be modified using function calls that only set the hardware interrupt enable bit when both the interrupt scheduler and user code want to do so. This is simple to implement but adds time and space overhead.

Implications of dropping interrupts A potential problem with scheduling interrupts is that during overload the CPU may miss some interrupts that otherwise would have been processed. This could lead to degraded quality of service or even system failure. The rationale for using interrupt schedulers is as follows. When interrupt overload occurs there is no way to avoid making a difficult tradeoff—either interrupts must be dropped or else other processing will starve. Our work makes an implicit assumption that a system's core processing is more important than, for example, receiving every network packet that arrives. Clearly this assumption could be incorrect for a particular system. In general, however, we strongly belive that failures should be forced to occur in a predictable, bounded manner, with as little impact on the rest of the system as possible. Interrupt schedulers can help achieve this goal.

5. Performance Analysis

Interrupt controllers implicitly run interrupts at a higher priority than non-interrupt work. It is well-known that static priority schedulers have poor fairness characteristics during overload: low priority work is starved [21]. The goal of our work is to avoid this kind of starvation by applying reservation-like scheduling techniques to interrupts. In this section we show how to make quantitative performance guarantees to low-priority work in the presence of scheduled interrupts—this cannot be done otherwise, except by making risky assumptions about maximum interrupt arrival rates.

Parameter	*Cost (cycles)*
t_{int}	79
t_{poll}	4
t_{setup}	5
t_{expire}	79
t_{flip}	5
t_{count}	12
t_{clear}	5

Table 2. Overhead constants for the ATmega103L with TinyOS. A cycle is 250 ns.

5.1 Static priority analysis

There are many different priority-based real-time analyses [8, 16, 26, 29, 25]. The common idea across all of this work is that given a worst-case execution time (WCET), a minimum interarrival time, and a priority for each member of a collection of tasks, the worst-case completion time of each task instance, relative to the time it became ready, can be efficiently computed. In the next section we show how to compute WCET (denoted C) and minimum interarrival time (denoted T) for each interrupt in an embedded system. Our work does not address the problem of computing the WCET of generic code; this is a well-studied static analysis problem [6, 17]. Rather, given some basic system overheads and a WCET for the user-specified part of the interrupt, we show how to put these numbers together into an aggregate WCET that includes all overheads. Once C and T have been computed for all tasks, an appropriate real-time analysis can be run to find out if a system is schedulable. The details of the analysis chosen are irrelevant: we simply focus on deriving inputs that are common across real-time analyses.

5.2 Modeling interrupt schedulers

Consider a system with a single interrupt handler that is connected to an external device. We want to ensure that every pair of interrupts processed by the CPU is separated by at least the minimum interarrival time t_{arrival}. Let t_{work} be the worst-case execution time of processing a unit of work generated by the device, t_{int} be the overhead of taking an interrupt as opposed to polling (usually just the cost of the interrupt prologue and epilogue), and t_{poll} be the cost of polling: determining if the device has any new work that needs processing. Furthermore, let t_{setup} be the time taken to arrange for a one-shot timer interrupt to arrive in the future and t_{expire} be the overhead to take either a periodic or one-shot timer interrupt. For the bursty scheduler, let t_{count} be the overhead of incrementing the interrupt counter and checking it against the threshold value, and let t_{clear} be the cost of clearing this counter. Finally, let t_{flip} be the overhead for either setting or clearing an interrupt enable flag. Of these overheads, it is usually the case that only t_{work} is under control of the developer—the other constants are determined by the platform: the hardware and RTOS. For example, the values of these constants on our test platform (described in Section 6) are given in Table 2. We computed these values empirically by counting instructions; they are approximate.

The rest of this section shows how to compute the important real-time parameters C and T for each interrupt source.

Pure interrupts In the worst case, interarrival time of interrupts is zero, and low-priority work is starved. This condition corresponds to a stuck level-triggered interrupt.

Pure polling Polling is driven by a timer that expires once every minimum interarrival time, and in the worst case work from the device must be processed at each expiration. This situation can be modeled as a periodic task with $T = t_{\text{arrival}}$ and $C = t_{\text{expire}} + t_{\text{poll}} + t_{\text{work}}$.

Strict software scheduler This scheduler can be modeled as a pair of tasks, one representing the interrupt handler, the other representing the timer interrupt that re-enables the device interrupt, both with $T = t_{\text{arrival}}$. To see that this is correct, first notice that since the timer interrupt is always set to expire one interarrival time in the future, it cannot recur more often than this. Second, the interrupt itself cannot recur more often than once every t_{arrival} because each time it arrives, its enable bit is cleared for one interarrival time. The worst-case execution times are as follows: for the interrupt $C = t_{\text{int}} + t_{\text{flip}} + t_{\text{setup}} + t_{\text{work}}$, and for the timer $C = t_{\text{expire}} + t_{\text{flip}}$.

Bursty software scheduler Again, we model the interrupt and timer tasks separately. The burst size N can take any value, and the period T of the timer and interrupt are both equal to minimum interarrival time for bursts of interrupts. Then, for the timer, $C = t_{\text{expire}} + t_{\text{clear}} + t_{\text{flip}}$ and for the interrupt $C = N(t_{\text{int}} + t_{\text{work}} + t_{\text{count}}) + t_{\text{flip}}$. In other words, for purposes of real-time analysis, we model a worst-case burst of interrupts as a single task arrival.

When using this scheduler it is possible for a burst of interrupts to arrive just before the periodic timer interrupt, and then for a second burst to arrive immediately after. Many real-time analyses can deal correctly with this case: the key is to avoid making any assumption about when, within its period, a task will run; this is handed by a *release jitter* term in the schedulability equations [29]. The jitter for a burst of interrupts should be set to $T - C$. An alternate approach, also correct but more pessimistic, is to double the WCET of the task representing the burst of interrupts.

Hardware scheduler The interrupt scheduler permits at most one interrupt per interarrival time, and therefore it can be modeled as a periodic task with $T = t_{\text{arrival}}$ and $C = t_{\text{int}} + t_{\text{work}}$.

6. Experimental Evaluation

The analytical results in the previous section can be used to compute lower bounds on the rate of progress of low-priority work in an embedded system. These bounds are best computed using scheduling theory as they are not easy to determine empirically. This section uses experiments run on a real system—no simulation results are used—to show that our techniques work and to evaluate their overhead in practice. In each case, our hardware-based interrupt scheduler that is implemented on a second microcontroller represents the "gold standard" against which the software schedulers should be compared: it provides perfect protection with zero software overhead.

6.1 Methodology and equipment

To evaluate our three interrupt schedulers, we implemented the software schedulers on Berkeley "Mica" motes [10], sensor network nodes based on Atmel's ATmega103L microcontroller. These processors run at 4 MHz and have 4 KB of SRAM for data storage. Because of their small size, they almost always run only one application, allowing the application to have full control over the interrupt arrival rate restrictions. Our prototype hardware scheduler is implemented as a special-purpose program running on a second microcontroller.

In the experiments in Sections 6.2 and 6.3, the mote was presented with externally generated periodic interrupts at frequencies

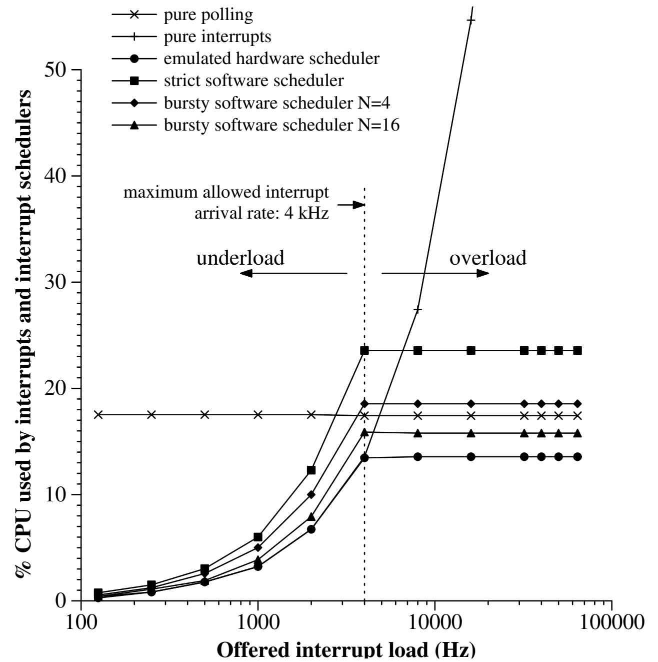

Figure 1. Comparing the performance of different interrupt schedulers when interrupt handlers perform no work

between 0.26 kHz and 16 kHz. Interrupt schedulers were set to enforce a maximum arrival rate of 4 kHz. We inferred the CPU overhead of scheduling and handling the interrupts by observing the rate of progress of a background task running on the mote. There was very little variation across repetitions of the experiments and so we omit confidence intervals.

Although 16 kHz is a high frequency, it is not uncommon for embedded systems, especially those connected to "dumb" hardware, to deal with lots of interrupts, as indicated in Table 1. Also, for example, the TinyOS motes, during the start symbol detection phase of wireless radio communication, take interrupts every 50 μs, a 20 kHz arrival rate.

6.2 Overhead of scheduling interrupts

In our first experiment, the interrupt handler returns without performing any real work. This, coupled with the high maximum interrupt rate, was designed to avoid masking any overheads associated with our interrupt scheduling techniques. The results of this experiment are shown in Figure 1. The "pure polling" and "pure interrupts" lines were the controls in this experiment, and their overheads are as expected: polling has constant overhead that is independent of the interrupt arrival rate, while the overhead of handling interrupts in the standard way is linear in the interrupt arrival rate.

In contrast with polling, all of our interrupt scheduling techniques approach zero CPU overhead when interrupts are infrequent. In contrast with interrupts, the overhead of all of our techniques flattens out even in the presence of very high frequency interrupts. Thus, all of our schemes achieve our goals of low overhead in the expected case while avoiding CPU overload under high interrupt loads.

The interrupt schedulers implemented in software incur some overhead. Figure 1 shows that each of the software schedulers approximates the ideal performance of the hardware interrupt scheduler more or less closely. In terms of lost CPU capacity relative to the hardware interrupt scheduler, the strict software scheduler

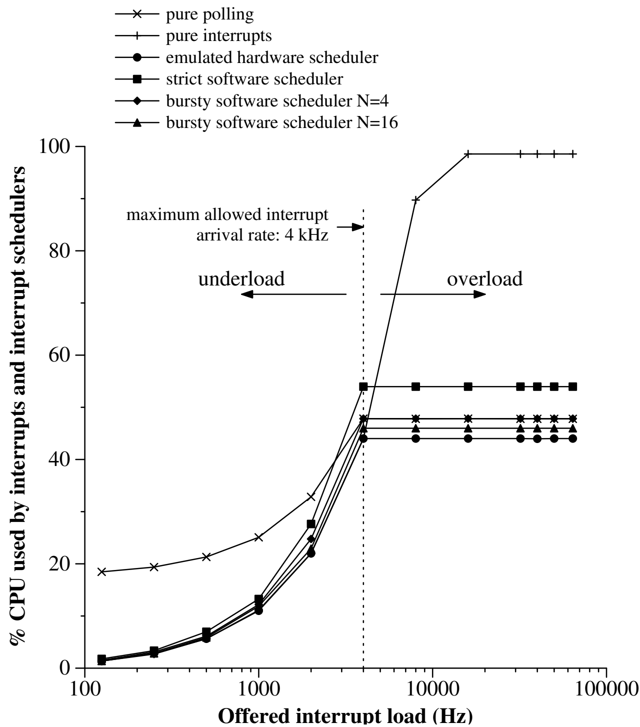

Figure 2. Comparing the performance of different interrupt schedulers when interrupt handlers perform 250 cycles (62.5 μs) of work

has at most 10% overhead, the bursty scheduler with $N = 4$ has at most 5.0% overhead, and the bursty scheduler with $N = 16$ has at most 2.2% overhead. The hardware interrupt scheduler, as expected, incurs no software overhead: its performance is indistinguishable from pure interrupts when the system is underloaded, and its CPU load is perfectly flat during overload.

6.3 Scheduling interrupts that perform work

Above, we examined the performance of interrupt scheduling techniques in the extreme case where the interrupt handler does not do any real work. While this helps us clearly identify the performance strengths and limitations of each scheduling technique by exaggerating its overhead, it is not realistic. In Figure 2 we present the results of a similar experiment where the interrupt handlers perform 250 cycles of busy-work. We chose 250 cycles because measurements of two simple but representative TinyOS kernels, CntToLedsAndRfm and RfmToLeds, showed that they spent approximately this much time handling each interrupt, on average.

Figure 2 shows that the performance penalty for limiting interrupt arrival rates using software-based schedulers is relatively small when the interrupt handler performs a realistic amount of work. Note that the "pure polling" data points in this figure, unlike Figure 1, show CPU utilization that increases with offered interrupt load. This happens because when there are few interrupt arrivals, the polling task has little work to do; when there are many interrupt arrivals, it is frequently or always forced to spend 250 cycles performing work.

6.4 Scheduling multiple interrupt sources

We performed an experiment to look at the effect of running multiple interrupt schedulers in the same system, to ensure that they compose properly and to investigate amortizing scheduling overhead across several interrupt sources.

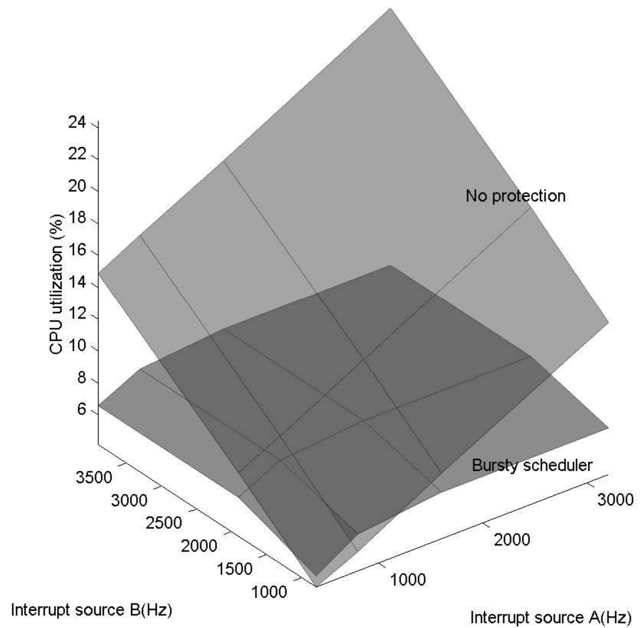

Figure 3. Aggregate CPU usage of two interrupt sources with and without overload protection

Figure 3 shows the aggregate CPU utilization of two interrupt sources, first without any protection, then with bursty interrupt schedulers that share a common 200 Hz timer interrupt for clearing their burst counts. The maximum burst size for source A is five interrupts and the maximum burst size of source B is seven, resulting in maximum average arrival rates of 1 kHz and 1.4 kHz, respectively.

By comparing CPU utilizations before and after adding the schedulers, we can estimate their aggregate overhead. When the arrival rates of sources A and B are 400 and 781 Hz, respectively, the difference in utilization between the scheduled and unscheduled systems is 1.1%. In contrast, adding a bursty scheduler to a system with a single interrupt source results in 4.1% overhead when the burst size is 4, and 2.1% overhead when the burst size is 16. These measurements were taken for a system with a single interrupt source running at 1 kHz, which is slightly less than the combined frequencies of the two-source system. The two-source system is more efficient because the overhead of the timer is amortized across the two interrupt sources. Extrapolating these results, we believe that adding a third interrupt source would result in less than 1% additional overhead.

7. Case Study: Protecting Against Network Overload

As a further demonstration of the utility of interrupt overload protection, we implemented an interrupt scheduler on an Ethernut [7] node. The Ethernut board version 1.3f is an embedded device containing a 10 Mbps Ethernet interface and an AVR ATmega128 processor running at 16 MHz. The Nut/OS software for Ethernut boards provides basic RTOS services such as threads and timers, in addition to a full TCP/IP stack with associated protocols such as DHCP. Figure 4 shows our experimental setup.

We started with an example Ethernut application that provides a web server, and we added functionality that plays a tone by bit-banging a digital output line that we connected to a speaker. The

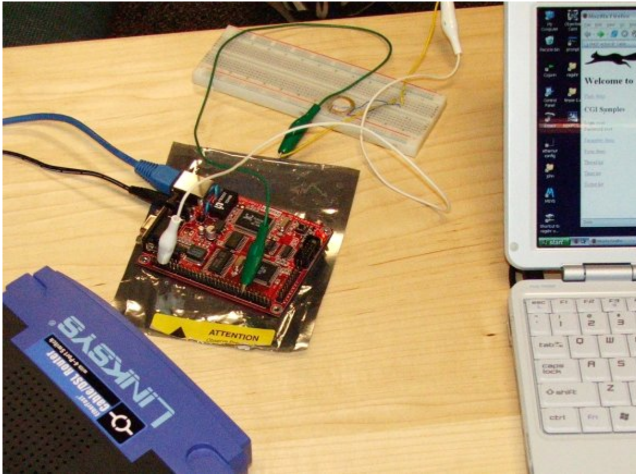

Figure 4. Experimental setup for Section 7. The Ethernut board (center) drives a piezo speaker on a breadboard (top), and is connected to a laptop through an Ethernet switch.

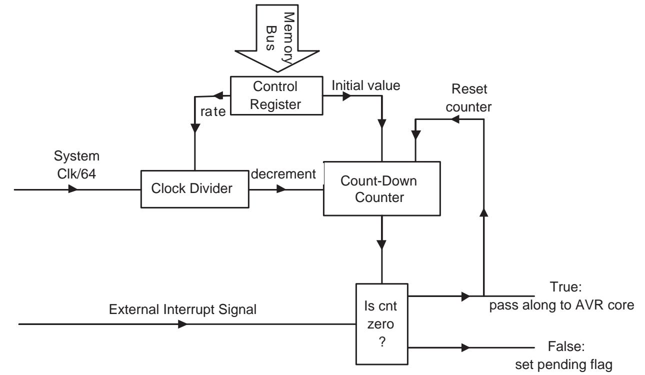

Figure 5. Schematic view of the hardware interrupt scheduler added to an AVR-like core

Parameter	*Original*	*Modified*	*Change*
lines of VHDL code	5 847	5 951	1.8%
4-LUTs used	2 966	2 983	0.6%
cells used	3 468	3 527	1.7%
max. frequency	37.6 MHz	36.3 MHz	-3.5%

Table 3. Comparison of the original OpenCores AVR and one augmented with an interrupt scheduler

speaker-driving code ran in thread context at maximum priority, using an Ethernut timer to schedule itself every 2 ms. Each time it ran, it flipped the sense of the output wire; the resulting square-wave produced a 250 Hz tone.

Next, we subjected the Ethernut board to increasing packet loads. The packets were minimum-sized Ethernet frames sent by a PC. At a few hundred packets per second (PPS) the tone being played by the Ethernut board was audibly distorted, and finally between 1500 and 1600 PPS interrupt overload occurred: the board stopped producing sound for the duration of the packet flood.

We added a bursty interrupt scheduler to limit the Ethernet interrupts to at most a burst of 15 interrupts every 25 ms, corresponding to a maximum packet arrival rate of 600 PPS. The choice of 15 for burst size was domain-specific: on this platform a typical small web transaction generates 14 interrupts, and it did not seem to make sense to consider overload to be occurring while processing a single web request. With the interrupt scheduler enabled, the board continues to produce a tone even when subjected to heavy packet loads of more than 10,000 PPS. Finally, note that the 10 Mbps NIC is quite slow—more recent versions of the Ethernut hardware contain a 100 Mbps interface that can overwhelm the AVR processor with interrupts when receiving only a small fraction of its maximum packet load.

8. Interrupt Overload Protection in Hardware

In Section 6 we evaluated a prototype hardware interrupt scheduler implemented using a separate microcontroller. Our second prototype interrupt scheduler is implemented in VLSI logic. We started with a design, in VHDL, for an AVR-like processor provided by OpenCores [22], a collection of free hardware design files. The core is not an authentic Atmel core, but it implements the same architecture and instruction set as the ATmega103, the chip used in the Berkeley Mica motes. It is missing some features found on the real Atmel chips, but none that we needed.

We added logic implementing the functionality described in Section 3.4: a memory-mapped control register for the interrupt scheduler, an internal count-down register, and associated control logic. The maximum allowed rate can be set to any of the 256 non-uniformly distributed values in the range of 500 Hz and 256 kHz. Figure 5 shows a high-level schematic view of the added logic.

Adding an interrupt scheduler to the AVR-like core required adding about 100 lines of VHDL. We synthesized the original OpenCores AVR and our augmented AVR for the Spartan-3 XC3S400, a 400,000-gate FPGA. Some statistics about the two designs are shown in Table 3. A 4-LUT is a basic building block that can implement any four-input Boolean function and a cell is a generic term for all basic building blocks, including 4-LUTs as well as other digital gates. We found that adding a hardware interrupt scheduler for a single external interrupt line increases the number of logic units used by 1.7%, or approximately 600 transistors. These costs are modest and also reflect an unoptimized implementation; we believe they could be reduced.

We tested the hardware interrupt scheduler in simulation and verified that the hardware-based scheduler behaves correctly. The simulation was done at the behavioral VHDL level. Direct performance comparisons between this hardware scheduler and the schedulers that we evaluated in Section 6 would not be meaningful—the chips have different pipelines and, hence, different performance characteristics.

Many embedded developers do not have the luxury of doing VLSI design as part of creating an embedded system. However, FPGA-based systems are growing in popularity, as are system-on-chip and network-on-chip designs, which emphasize parameterized reuse of existing designs, in order to create embedded computers that provide exactly the right amount of functionality for a particular application, minimizing waste. Our hardware interrupt scheduler could be added to one of these designs in a straightforward way. Furthermore, there is increasing interest in hybrid platforms such as Atmel's family of AT94S devices, which combine an AVR processor and a small FPGA in a single package. Chips in this family cost only a few dollars; they would make an ideal platform for our hardware interrupt scheduler. Finally, we believe that hardware vendors could add interrupt schedulers to embedded processors at negligible cost, making it easy for users of these platforms to create software with strong protection from interrupt overload.

9. Related Work

Receive livelock is the condition where a network server is overwhelmed by arriving packets and spends most or all of its time processing interrupts. Mogul and Ramakrishnan [19] designed a network subsystem that switches from interrupt-driven to polling when the system appears to be overloaded. Similarly, lazy receiver processing [5] provides early demultiplexing of network traffic and proper accounting of time spent processing it. These efforts focus on maximizing throughput and on achieving long-term fairness, without making any specific guarantees about timely execution to applications. On the other hand, we focus on predictability and responsiveness, making strong performance guarantees to non-interrupt work. Also, receive livelock solutions tend at be network-specific, for example inferring livelock due to input queue overflow. Little has been done to address the more general interrupt overload problem.

Hardware assistance can help avoid interrupt overload. For example, interrupt mitigation techniques, such as those provided by the Intel 21143 Ethernet controller [3], can delay the onset of interrupt overload or prevent it. More flexible solutions are provided by lazy receiver processing [5] and Dannowski and Härtig's work [4], which modify NIC firmware to drop packets in order to prevent network interfaces from overloading hosts. Again, however, the focus is on maximizing throughput: the NIC's goal is to ensure that the host is keeping up, rather than ensuring that incoming flows respect a given maximum packet rate. These approaches could be modified to better support embedded concerns but even so, their applicability would be limited: most embedded peripherals do not have enough processing resources to run their own interrupt-limiting code. Our hardware interrupt scheduler, on the other hand, does not require smart peripherals; rather, a few hundred transistors worth of logic must be added to the main CPU.

QNX [9], TimeSys Linux [28], and a number of other systems, such as those we cited in Section 2, run as much "interrupt" code in thread mode as possible—the actual handler for each interrupt then becomes a stub that awakens the corresponding thread. This approach can delay, but not prevent, interrupt overload, unless two conditions are met. First, the interrupt must remain disabled until the thread has finished its processing. Second, the thread scheduler must not blindly give interrupt threads higher priority than non-interrupt work: it needs to use some sort of reservation-based scheduling policy. Nemesis is the only OS that we are aware of that meets both criteria. However, even when implemented properly, scheduling interrupts as threads increases overhead by adding context switches to the critical path for interrupt handling. This overhead may be acceptable on fast systems, but thread dispatch is relatively expensive on small embedded processors. Furthermore, many small embedded devices, such as the TinyOS motes [10], are so resource-limited that they do not even have a thread scheduler.

The operating systems and real-time communities have produced many results on scheduling strategies that provide isolation between concurrent tasks, protecting against tasks that arrive too often or run for too long. These results include aperiodic servers [27], processor reservations [12, 18], and rate-based scheduling techniques [11]. While these results are useful, our work is different in that we are focusing on low-level scheduling mechanisms that can efficiently throttle interrupt arrivals, rather than focusing on high-level scheduling policies at the level of threads or processes. It is not straightforward to use a standard scheduling algorithm to schedule interrupts because interrupt arrivals are effectively scheduled in hardware by the interrupt controller, out of control of systems software.

Abeni and Lipari [1] and Regehr and Stankovic [24] have investigated adaptive schemes that compensate user-level tasks for CPU time that is "stolen" from them by interrupts. However, neither of these solutions throttles interrupt arrivals and so they will not work when interrupt load is too high.

The time-triggered architecture [14] advocates avoiding interrupts in favor of a polling-based approach to interaction with devices. Our work shows that interrupts need not introduce the possibility of starvation or unacceptable delays in embedded systems. There is no inherent problem with using rate-limited interrupts in safety critical systems.

Finally, in networked embedded systems the babbling idiot problem [2] occurs when a node begins sending too much traffic onto a network. Avoiding interrupt overload, then, is basically the inverse babbling idiot problem: protection is implemented on the input side rather than on the output side.

10. Conclusions

While the research community has spent a great deal of effort on providing processor isolation between threads and processes, little work has been done on providing strong performance guarantees to embedded software in the presence of interrupt overload. We have developed, implemented, and evaluated two software-based mechanisms for protecting embedded systems against interrupt overload. We have shown their worst-case overheads are modest, on the order of 2%–10% of lost processor capacity for a fairly high frequency interrupt source (4 kHz) on weak processors: 4 MHz Atmel AVRs. Furthermore, overhead is quite small when interrupts arrive infrequently. We have also implemented and evaluated two interrupt schedulers in hardware: one is implemented on a second microcontroller, the other is added to an existing processor by modifying its description in VHDL. The latter has small overhead in terms of chip area, and we believe it shows that embedded chip vendors could provide interrupt overload protection as an added feature at little extra cost. Both hardware solutions add zero overhead to software running on the main processor in all cases.

Designers of embedded systems usually have an intuitive idea about the maximum rate of interrupt requests to expect from each interrupt source. However, with no way to enforce these maximum rates, the specifications lack teeth. Our work can encourage developers to answer a more pointed question about each interrupt source: "Under what circumstances is it better to drop interrupt requests than to attempt to process them?" The resulting system can be tested at and above the maximum interrupt rate; we believe this will lead to more robust embedded systems based on more guarantees and less guesswork.

Acknowledgments

The authors would like to thank Luca Abeni, Ian Broster, Jay Lepreau, and Alastair Reid for their helpful comments on drafts of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 0209185.

References

[1] Luca Abeni and Giuseppe Lipari. Compensating for interrupt process times in real-time multimedia systems. In *Proc. of the 3rd Real-Time Linux Workshop, Work in Progress Session*, Milan, Italy, November 2001.

[2] Ian Broster and Alan Burns. An analysable bus-guardian for event-triggered communication. In *Proc. of the 24th IEEE Real-Time Systems Symp. (RTSS)*, pages 410–419, Cancun, Mexico, December 2003.

[3] Intel Corporation. 21143 PCI/CardBus 10/100Mb/s Ethernet LAN Controller, October 1998. `ftp://download.intel.com/design/network/manuals/27807401.pdf`.

[4] Uwe Dannowski and Hermann Härtig. Policing offloaded. In *Proc. of the 6th IEEE Real-Time Technology and Applications Symp. (RTAS)*, pages 218–227, Washington, DC, May 2000.

[5] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A network subsystem architecture for server systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, pages 261–276, Seattle, WA, October 1996.

[6] Jakob Engblom, Andreas Ermedahl, Mikael Nolin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *Journal of Software Tool and Transfer Technology (STTT)*, 4(4):437–455, August 2003.

[7] Ethernut: Tiny embedded Ethernet devices. `http://www.ethernut.de`.

[8] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Technical Report 2966, INRIA, Rocquencourt, France, September 1996.

[9] Dan Hildebrand. An architectural overview of QNX. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[10] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer 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, November 2000.

[11] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *Proc. of the 20th IEEE Real-Time Systems Symp. (RTSS)*, pages 304–314, Phoenix, AZ, December 1999.

[12] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, predictable scheduling of independent activities. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP)*, pages 198–211, Saint-Malô, France, October 1997.

[13] Scott Karlin and Larry Peterson. Maximum packet rates for full-duplex Ethernet. Technical Report TR-645-02, Princeton University, February 2002.

[14] Hermann Kopetz. The time-triggered model of computation. In *Proc. of the 19th IEEE Real-Time Systems Symp. (RTSS)*, pages 168–177, Madrid, Spain, December 1998.

[15] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[16] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[17] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, 17(2/3):183–207, November 1999.

[18] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. In *Proc. of the IEEE Intl. Conf. on Multimedia Computing and Systems*, May 1994.

[19] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[20] Charles Murray and Catherine Bly Cox. *Apollo: The Race to the Moon*. Simon and Schuster, 1989.

[21] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *Proc. of the 4th Intl. Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, November 1993.

[22] OpenCores. `http://www.opencores.org`.

[23] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. of the 24th IEEE Real-Time Systems Symp. (RTSS)*, Cancun, Mexico, December 2003.

[24] John Regehr and John A. Stankovic. Augmented CPU reservations: Towards predictable execution on general-purpose operating systems. In *Proc. of the 7th IEEE Real-Time Technology and Applications Symp. (RTAS)*, pages 141–148, Taipei, Taiwan, May 2001.

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

[26] Lui Sha, Ragunathan Rajkumar, and Shirish S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proc. of the IEEE*, 82(1):68–82, January 1994.

[27] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, June 1989.

[28] TimeSys Corporation. TimeSys Linux. `http://timesys.com/`.

[29] Ken Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems Journal*, 6(2):133–151, March 1994.