Eliminating Interrupt Overload in Embedded Systems

John Regehr *Usit Duongsaa*
School of Computing, University of Utah

Unpublished draft — May 2004

Abstract

Embedded systems can fail to operate correctly due to interrupt overload: starvation caused by too many interrupt requests. 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 solutions to the interrupt overload problem have several desirable properties. During underload, overhead is proportional to interrupt load. During overload, both the amount of work performed in interrupt context and its granularity are bounded, making it possible to provide strong progress guarantees to low-priority interrupts and non-interrupt tasks. These guarantees permit developers to avoid making assumptions about the worst-case interrupt rates of peripherals such as sensors and network interfaces, filling an important gap in the chain of reasoning leading to a validated embedded system. We show that our solutions successfully prevent interrupt overload with modest overhead by evaluating them 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.

1 Introduction

Most microprocessors manufactured in recent years were used in embedded applications [37]. These processors represent an important domain, running a variety of ubiquitous and useful applications, including many that are safety-critical such as medical automation and vehicle control. Use of embedded processors is growing rapidly. For example, a few decades ago cars contained no embedded processors; today a BMW 745i contains 71 microprocessors [18]; over the next few years technologies such as active collision avoidance, steer-by-wire, and variable valve timing will require even more computer power [38].

A typical embedded processor is connected to a number of external devices such as sensors, actuators, storage, and network interfaces. To avoid polling, many devices deliver work to the processor using interrupts. Work done in interrupt context is implicitly given a higher priority than work done in other contexts, making it dangerous to accept interrupts at an externally determined rate. However, this is precisely what interrupt-driven embedded systems do, making it possible for interrupts to be signaled often enough that *interrupt overload* occurs: the processor is forced to spend so much time handling interrupts that other tasks are delayed unacceptably.

Our goal was to devise techniques for eliminating interrupt overload 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 experienced by low-priority interrupts and non-interrupt work, in order to give them performance guarantees.
- It should be possible to make flexible tradeoffs between latency and throughput.
- Overhead must be reasonable.

Interrupt schedulers meeting these goals could be implemented in software or hardware; we explore both strategies. The main implementation challenge for a software-based scheduler is to efficiently regain control of interrupts from the hardware interrupt controller that gives interrupts higher priority than non-interrupt work. A static priority scheduler is unacceptable during overload because it starves low-priority work.

Our approach to eliminating interrupt overload is motivated by three main differences between embedded systems and general-purpose computer 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 [16, 19, 24]. Also, thread-level reservations cannot directly prevent interrupt overload, though they can delay its onset if "interrupt" code 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 [30] we attempted to run some TinyOS [13] 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. 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-driven network subsystems [8, 25] 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 [8] and Dannowski and Härtig [7] have prevented interrupt overload by changing the firmware running on high-end NICs.

During the course of this work we were surprised in two ways. First, we found that interrupt overload can be avoided with modest overhead, even when making strong guarantees to low-priority work in the presence of interrupts with kilohertz arrival rates—all of this on 4 MHz processors. Similarly, our hardware-based solution to interrupt overload has little overhead in terms of chip area. Second, while there is a great deal of related work, we are not aware of any solutions to the specific problem of avoiding interrupt overload in embedded software.

In the next section we describe potential causes for interrupt overload. We present our solutions to interrupt overload in Section 3, analyze their behavior in Section 5, and experimentally validate and evaluate them in Section 6. Section 7 describes an experiment in synthesizing a hardware-based interrupt scheduler. We compare our work to related research in Section 8 and conclude in Section 9.

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 [26, 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 economic—embedded systems are usually very cost sensitive: unit cost must be minimized. 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?

The data in Table 1 shows some measured and computed worst-case interrupt rates. Even innocuous-seeming hard-

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
CAN bus	15 000
10 Mbps Ethernet	19 500
I2C bus	50 000
USB	90 000
100 Mbps Ethernet	195 000
Gigabit Ethernet	1 950 000

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.

ware, 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.

Our work provides a way to prevent high-frequency transient effects from reaching an embedded processor, using either software or hardware to limit the rate of interrupt arrivals. The traditional way to debounce a switch, on the other hand, is to add analog or digital hardware implementing a low-pass filter. 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.

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 (64-byte) packets, the interrupt rate increases to more than 19 kHz, potentially starving important processing.

Finally, support for CPU reservations is increasingly popular in real-time operating systems (RTOSs). For example, TimeSys Linux [35], LynuxWorks LynxOS-178 [22], WindRiver VxWorks AE653 [39], and Green Hills Software Integrity-178B [10] all provide strong isolation between processes. The idea is that if the RTOS can enforce non-interference between workloads, then workloads with widely varying levels of criticality can share a single processor safely. This is expected to save considerable hardware cost through multiplexing, and considerable development time by eliminating the need to certify less critical software at the same level as more critical software. Although these operating systems attempt to run as much interrupt load in schedulable contexts as possible, they are still vulnerable to interrupt overload, based on our interpretation of published descriptions of these systems. The possibility of unanticipated crosstalk between separate applications running on a single processor, some safety-critical, should be considered to be a major design flaw.

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. For code written in high-level languages, the compiler or runtime library supplies an interrupt *prologue* and *epilogue*

that respectively run before and after any user-supplied interrupt code. Their purpose is to save and restore the machine state, preventing interrupt code written by developers from corrupting the register file or condition codes.

On the AVR, even though lower-numbered interrupts get higher priority according to the interrupt arbitration logic, a higher-numbered pending interrupt will still preempt a low-numbered running interrupt provided that its enable bit, and the global enable bit, are set. In contrast, other processors such as the x86 have an *interrupt request level* that always prevents a high-numbered interrupt from preempting a low-numbered interrupt.

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. 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 before the processor's interrupt arbitration logic while the latter is logically after 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, must be 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.

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 not often exceeded. This trick is only applicable 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 pathological case where three back-to-back bursts of device interrupts can 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. When burst size is one, this scheduler does not quite degenerate to the strict scheduler: since the timer is asynchronous it may be out of phase with interrupt arrivals, while the strict scheduler's timer never is.

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 serious 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 associates two special registers with an interrupt source. The first is a counter that is auto-

matically decremented at a high frequency. When an interrupt arrives, there are two possibilities:

1. The counter is at zero. In this case the counter is set to an initial value that is stored in the second special register. The initial value register must be writable from software. Then, the interrupt signal is propagated to the CPU and the counter starts counting down.

2. The counter is not at zero. In this case the interrupt arrival is noted and no other action is taken. When the counter reaches zero, the interrupt is handed to the CPU and the counter is reset so it can begin counting down again. 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.

The software overhead of this solution is zero: enforcement of minimum interarrival times is completely free. We will show in Section 7 that implementing a scheduler in hardware is cheap in terms of chip area.

3.5 Scheduling multiple interrupt sources

Scheduling multiple interrupt sources using the hardware scheduler is trivial: the hardware is replicated once per interrupt line. Similarly, the strict software scheduler is easily replicated given a virtual timer library implemented in software, although this increases overhead when compared to direct interaction with a hardware timer. 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 poor overall performance. For example, consider a system with three interrupt sources whose maximum burst arrival rates are 324, 200, and 757 Hz. The LCM of these frequencies is 12263400. Running a periodic timer at more than 12 MHz is likely to choke an embedded processor. A better solution would be to round the maximum burst frequencies up slightly, for example to 330, 220, and 770 Hz, so that a periodic timer running at 110 Hz could clear the interrupt counters on every third, second, and seventh invocation, respectively. Even more timer interrupts are eliminated if we can afford to relax the maximum burst arrival rates to 400, 200, and 800 Hz, and then run the periodic timer at 200 Hz. Interrupt rate limits are likely to be adjustable, since they are designed to stop exceptional behavior, not deal with the expected case. 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 implementing a bursty scheduler in hardware. 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 by doing this. Other parts of the design space, such as attempting to improve expected-case performance using soft timers [2], would be worth exploring but are beyond the scope of this paper.

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.

Losing interrupts A potential problem with scheduling interrupts—in either hardware or software—is that we might cause the CPU to miss some interrupt signals that it would otherwise have processed. There are two responses to this criticism. First, the tradeoff here is an unavoidable one: the CPU cannot be guaranteed to process all interrupts, regardless of arrival rate, and still make timeliness guarantees to other activities in the system. Second, by using appropriate real-time analyses, developers can ensure that interrupts are never missed when all interrupt sources behave as expected.

The penalty for missing an interrupt request is device-specific. For example, missed timer interrupts are a problem

since they can skew a system's notion of time. Network interrupts, on the other hand, can often be harmlessly dropped as long as the interrupt handler is written to drain the entire input queue, rather than processing a single packet per interrupt.

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 [27]. 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.

5.1 Static priority analysis

Starting with Liu and Layland's 1973 paper on rate-monotonic analysis [20], the theory and practice of preemptive priority-based scheduling has been worked out in great detail, culminating with generalized rate monotonic analysis [33] and static priority analysis [36] during the 1990s. Although there are many variations, 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. A task instance is just an invocation of a task, such as an interrupt request or a web request. If the completion time of each task is smaller than some domain-specific deadline, then a system is said to be *schedulable*.

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 [5, 9, 21]. 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. Then, in Section 5.3 we provide an example of making timeliness guarantees to low-priority work, and in Section 5.4 we discuss the general problem of accurately modeling interrupt subsystems using real-time equations.

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.

5.2 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 Since we are trying to enforce, rather than assume, that interrupts respect a minimum interarrival time, we must consider the minimum interarrival time of unscheduled interrupts to be zero, making it impossible to make progress guarantees to lower-priority work.

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 entity.

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 [36]. 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 $C = t_{\text{int}} + t_{\text{work}}$.

5.3 An example

Consider an embedded system whose performance characteristics match those given in Table 2. The processor is used to run a feedback loop that must execute at 250 Hz, or every 4 ms. Furthermore, in order to maintain stable control, each invocation of the feedback loop must complete before the next is ready to run. The feedback loop runs in the non-interrupt context of the processor and requires 2 ms of CPU time per invocation. The processor is connected to a control network, a 10 Mbps Ethernet. The network interface interrupts on packet arrival, and the network interrupt handler requires 0.15 ms to complete in the worst case.

A back-of-the-envelope analysis shows that if a malicious or malfunctioning node sends back-to-back minimum sized packets, the resulting 19500 Hz interrupt signal will overload the CPU, severely starving the feedback loop. To avoid this problem, we will run a bursty interrupt scheduler that permits a burst of 15 network interrupts every 10 ms, for a maximum average interrupt (and packet) processing rate of 1500 Hz. Cast in terms of real-time analysis, using the rules for the bursty scheduler from the previous section, this gives the task set shown in Table 3.

Task	Priority	Period (T)	WCET (C)
periodic timer	high	10 ms	0.022 ms
network intr. burst	high	10 ms	2.592 ms
feedback loop	low	4 ms	2.000 ms

Table 3: Concurrent tasks modeling the example system described in Section 5.3 with a scheduler permitting a burst of 15 network interrupts every 10 ms

Task	Priority	Period (T)	WCET (C)
periodic timer	high	10 ms	0.022 ms
network intr. burst	high	2 ms	0.520 ms
feedback loop	low	4 ms	2.000 ms

Table 4: Concurrent tasks modeling the example system described in Section 5.3 with a scheduler permitting a burst of 3 network interrupts every 2 ms

Running this task set through a schedulability analysis shows that the interrupt scheduler does not provide enough isolation: an invocation of the feedback loop may complete as late as 7.2 ms after it becomes ready to run—3.2 ms after the next invocation should have started to run. The intuition, in this case, is simple: at the same time that an invocation of the feedback loop starts to run, a burst of 15 network interrupts arrives, a timer arrives to reset the burst counter, and then a second burst of 15 network interrupts arrives. Then, finally, the invocation of the feedback loop runs to completion. Since each interrupt burst runs for about 2.6 ms, the elapsed time to completion of the feedback loop is 7.2 ms. In more complex situations, manual reasoning about task completion times is impossible. The real-time analysis that we used is a standard preemptive static-priority analysis; it is one of several provided by SPAK [29]. We confirmed the analytic results using the simulator included with SPAK.

To make the system schedulable without decreasing the number of network packets the system will accept, we can modify the bursty scheduler to accept a burst of three interrupts every 2 ms, leading to the task set shown in Table 4. Re-running the real-time analysis shows that the feedback loop is now guaranteed to finish running less than 3.6 ms after it starts, leaving a comfortable safety margin before the 4 ms deadline. Since the interrupt frequencies in this example are relatively low, decreasing the granularity of interrupt scheduling does not add much overhead: the utilization (average CPU load) of both task sets rounds to 76%.

This relatively simple example illustrates two points. First, by combining protection from interrupt overload with real-time analysis techniques, we can make strong performance guarantees to low-priority work. Second, making an embedded system work is often not so much a matter of maximizing throughput, but of carefully balancing the requirements for high throughput and rapid response time. Of course, the same tradeoffs are present in general-purpose operating systems where they are both easier to deal with (since timeliness requirements are less strict) and more difficult (since workloads are very difficult to predict and quantify).

5.4 Modeling interrupt subsystems

It has often been assumed (e.g., in [15] and [23]) that it is straightforward to model interrupt subsystems using standard static priority analyses. This view is somewhat optimistic, for several reasons:

1. The minimum interarrival time for externally generated interrupts often rests on shaky assumptions—this is the problem we address in this paper.
2. The interrupt controller on many microcontrollers cannot be modeled as a static priority scheduler. For example, although the AVR processors that we use for our performance evaluation (Section 6) use a hardware-based priority scheduler to select a pending interrupt to fire, the overall interrupt scheduling policy is *not* static priority preemptive scheduling because priorities apply only to pending interrupts, not those that are already running. To create true static priority preemptive scheduling for interrupts on this platform, interrupt masks must be manipulated so that a higher-priority interrupt disables all lower-priority interrupts before setting the master interrupt enable bit. This is conceptually trivial but, even after looking at half a dozen AVR programs from different sources, we have not seen any system that actually implements it, perhaps because it adds overhead and requires developer effort. Rather, a common situation is for interrupts to exist in a kind of free-for-all environment where any enabled interrupt can preempt any running interrupt.
3. The interrupt structure of many real systems uses a mix of preemptive and non-preemptive scheduling, for example running some interrupts with the master interrupt bit set, and running some interrupts with this bit cleared. This situation cannot be accurately modeled using either preemptive or non-preemptive static priority analysis techniques. Fortunately, more recent work on *preemption threshold scheduling* [32] can be used to accurately model a mix of preemptive and non-preemptive scheduling: it is a promising technique for modeling interrupt subsystems. Our SPAK package supports preemption threshold analysis.
4. Interrupts sometimes have a complicated internal structure, for example, running with all interrupts disabled for part of their execution, then enabling selected other interrupts later on.

As we have indicated, problems 2 and 3 can be solved using known techniques. Problem 4 is a non-issue for many embedded systems because their interrupt handlers have no complex internal structure: they simply run to completion without enabling or disabling any other interrupts. For systems where there are clever interactions between interrupt handlers, Brylow and Palsberg [5] have directly attacked the timing analysis problem. However, they have not yet shown how to make progress guarantees to interrupts running below the highest priority.

In general, it is not too difficult to create systems software whose performance can be analyzed. However, modeling typical embedded systems as they currently exist is challenging.

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 [13], 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.

For the software schedulers, which require timers, we took over one of the hardware timers on the AVR processors; overheads would have been slightly higher if we had used a timer that was virtualized in software. 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 between 0.26 kHz and 16 kHz. Interrupt schedulers

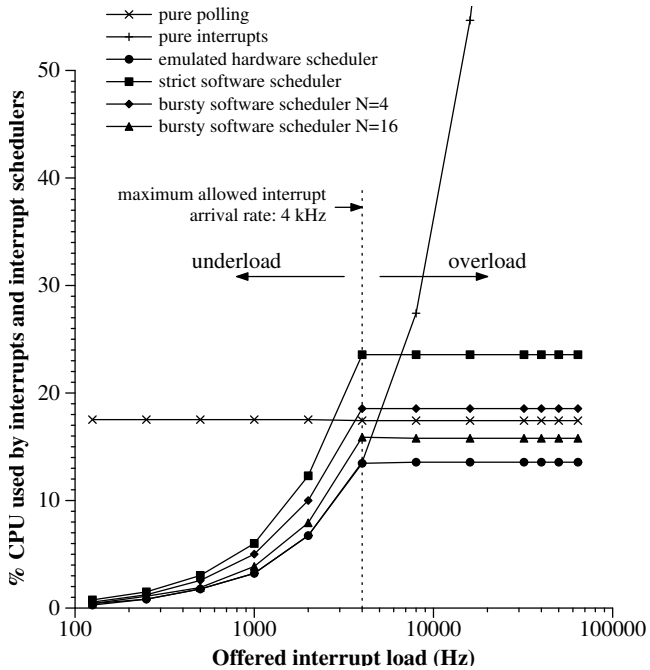

Figure 1: Comparing the performance of different interrupt schedulers when interrupt handlers perform no work

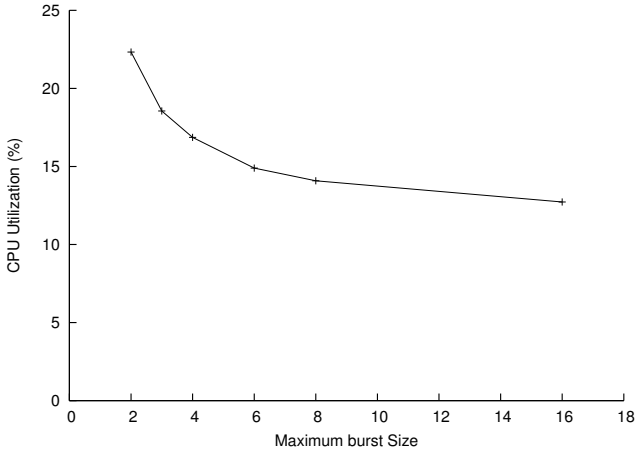

Figure 2: The effect of varying maximum burst size on the CPU utilization of an interrupt source. For all data points the offered interrupt load is 64 kHz and the interrupt handling rate is 4 kHz.

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 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.

Figure 2 shows the results of a different experiment. Its results indicate that the overhead of a bursty interrupt scheduler has two terms: one constant and one inversely proportional to burst size. The tradeoff is that as we showed in Section 5, longer bursts cause longer delays for non-interrupt activity.

6.3 Scheduling interrupts that perform work

Above, we examined the performance of each interrupt scheduling technique 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 3 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

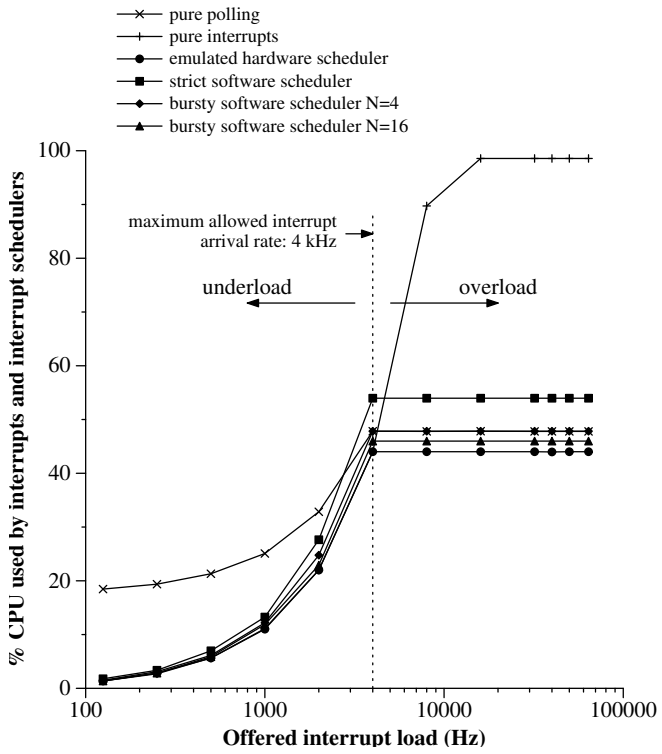

Figure 3: Comparing the performance of different interrupt schedulers when interrupt handlers perform 250 cycles (62.5 μs) of work

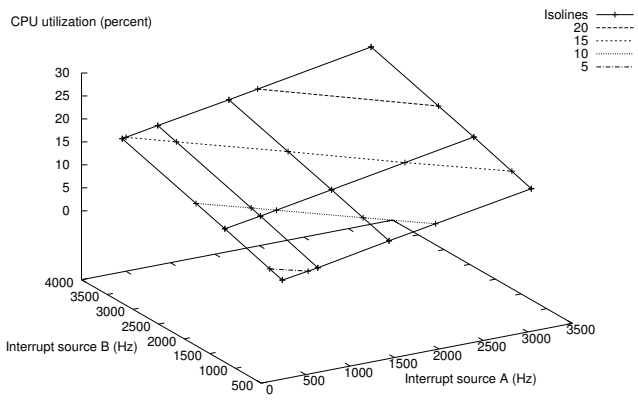

Figure 4: Aggregate CPU usage of two interrupt sources without protection from overload

two simple but representative TinyOS kernels, CntToLedsAndRfm and RfmToLeds, showed that they spent approximately this much time handling each interrupt, on average.

Figure 3 shows that adding some work to the interrupt handler does not affect the CPU capacity lost to scheduling overhead, but that the relative overheads of the schedulers starts to become negligible at low interrupt arrival rates. These results indicate that the performance penalty for limiting interrupt arrival rates using software-based schedulers is quite 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.

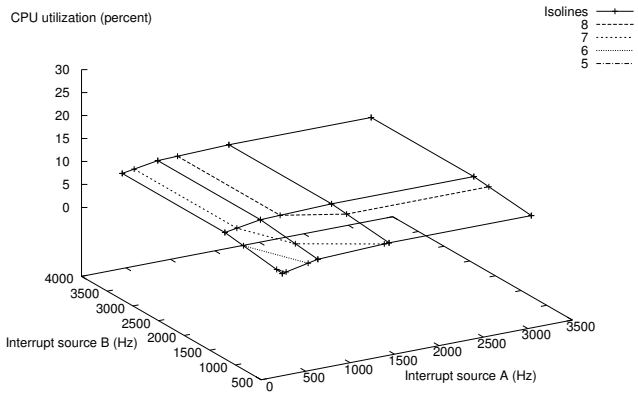

Figure 5: Aggregate CPU usage of two interrupt sources with bursty schedulers

Figure 4 shows the aggregate CPU utilization of two interrupt sources. Figure 5 shows the CPU utilization of the same two sources when throttled by 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% of additional overhead.

7 Adding Hardware Interrupt Overload Protection to a Microcontroller

In Section 6 we evaluated a prototype hardware interrupt scheduler implemented using a separate microcontroller. This section describes a second prototype interrupt scheduler, this one is implemented in VLSI logic. We started with a design, in VHDL, for an AVR-like processor provided by OpenCores [28], 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. Figure 6 shows a high-level schematic view of the added logic. In order to support a wide range of interrupt frequencies, the control register permits application code to specify both the initial value of the count-down register and also the rate at which it is decremented. The most significant bit of this register determines whether to count down at 64 kHz or 256 kHz, while the remaining seven bits store the value initially loaded into the count-down register when an interrupt is requested. This allows the interrupt arrival rate to be bounded at values between 500 Hz and 256 kHz.

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 5. 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.

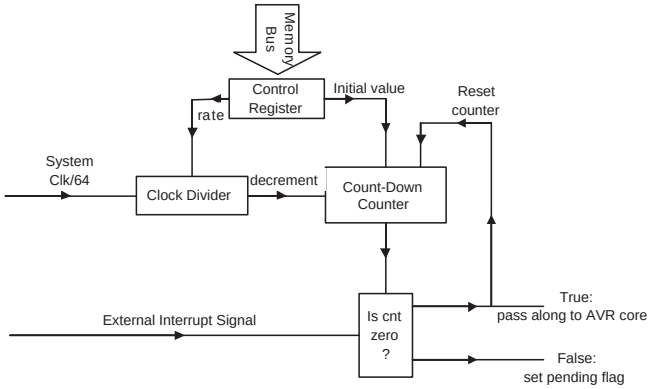

Figure 6: Schematic view of the hardware interrupt scheduler added to an AVR-like core

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 5: Comparison of the original OpenCores AVR and one augmented with an interrupt scheduler

We tested the hardware interrupt scheduler in simulation by generating signals above, below, and right at the maximum allowed frequency. By programming the core to set a bit every time it sees an interrupt, we verified that the hardware-based scheduler behaves correctly. The simulation was done at 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, minimiz-

ing 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.

8 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 [25] designed a network subsystem that switches from interrupt-driven to polling when the system appears to be overloaded. Similarly, lazy receiver processing [8] 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 [6], can delay the onset of interrupt overload or prevent it. More flexible solutions are provided by lazy receiver processing [8] and Dannowski and Härtig's work [7], 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.

L4 [11], Nemesis [19], QNX [12], TimeSys Linux [35], 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 [13], 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 [34], processor reservations [16, 24], and rate-based scheduling techniques [14]. 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 [31] 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 [17] 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 should be no problem with using rate-limited interrupts in safety critical systems.

Like our work, soft timers [2] help implementers make latency vs. throughput tradeoffs, and they mitigate the number of timer interrupts received by a system. It is possible that soft timers could be used in our software-based interrupt schedulers to improve their expected-case performance. However, soft timers are inherently optimistic, and would not improve worst-case performance.

Finally, in networked embedded systems the babbling idiot problem [4] 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.

9 Conclusions

Although much work has been done 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 should interrupt requests be ignored?" 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.

Our work has broader applicability than to sensor network nodes; the choice of TinyOS motes as an experimental platform was mainly a matter of expediency. At a low level these devices are representative of a large class of interrupt-driven embedded systems. However, in order to conserve power, sensor network nodes are expected to have low duty cycles and are therefore unlikely to suffer from interrupt overload. Even so, the motes are highly versatile and have even been used to control robots [3], an application where interrupt overload could easily occur.

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] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proc. of the 17th ACM Symp. on Operating Systems Principles (SOSP)*, pages 232–246, Kiawah Island, SC, December 1999.

[3] Sarah Bergbreiter and K. S. J. Pister. CotsBots: An off-the-shelf platform for distributed robotics. In *Intl. Conf. on Intelligent Robots and Systems (IROS)*, Las Vegas, NV, October 2003.

[4] 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.

[5] Dennis Brylow and Jens Palsberg. Deadline analysis of interrupt driven software. In *Proc. of the 11th Intl. Symp. on the Foundations of Software Engineering (FSE)*, pages 198–207, Helsinki, Finland, September 2003.

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

[7] 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.

[8] 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.

[9] 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.

[10] Green Hills Software. Integrity-178B RTOS. `http://www.ghs.com/products/safety_critical/integrity-do-178b.html`.

[11] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 66–77, Saint-Malô, France, October 1997.

[12] 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.

[13] 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.

[14] 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.

[15] Kevin Jeffay and Donald 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, December 1993.

[16] 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.

[17] 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.

[18] Stephen Lawton. Eternally yours at 8 bits. *Electronic Business*, October 2002.

[19] 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.

[20] 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.

[21] 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.

[22] LynuxWorks. LynxOS-178. `http://www.lynuxworks.com/rtos/lynxos-178.php3`.

[23] Jukka Mäki-Turja, Gerhard Fohler, and Kristian Sandström. Towards efficient analysis of interrupts in real-time systems. In *Proc. of the 11th Euromicro Workshop on Real-Time Systems*, York, England, May 1999.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] John Regehr. SPAK: a static priority analysis kit. `http://www.cs.utah.edu/~regehr/spak`.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

[37] Jim Turley. Embedded processors by the numbers. *Embedded Systems Programming*, May 1999.

[38] John F. West. Automotive engine control: where has it been, where is it going? *Embedded Control Europe*, November 2001.

[39] Wind River. VxWorks AE653. `http://windriver.com/`.