# Modeling device driver effects in real-time schedulability analysis: Study of a network driver [*]

Mark Lewandowski, Mark J. Stanovich, Theodore P. Baker, Kartik Gopalan, An-I Andy Wang
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530
e-mail: [lewandow, stanovic, baker, awang]@cs.fsu.edu, kartik@cs.binghamton.edu

## Abstract

*Device drivers are integral components of operating systems. The computational workloads imposed by device drivers tend to be aperiodic and unpredictable because they are triggered in response to events that occur in the device, and may arbitrarily block or preempt other time-critical tasks. This characteristic poses significant challenges in real-time systems, where schedulability analysis is essential to guarantee system-wide timing constraints. At the same time, device driver workloads cannot be ignored. Demand-based schedulability analysis is a technique that has been successful in validating the timing constraints in both single and multiprocessor systems. In this paper we present two approaches to demand-based schedulability analysis of systems that include device drivers. First, we derive load-bound functions using empirical measurement techniques. Second, we modify the scheduling of network device driver tasks in Linux to implement an algorithm for which a load-bound function can be derived analytically. We demonstrate the practicality of our approach through detailed experiments with a network device under Linux. Our results show that, even though the network device driver does not conform to conventional periodic or sporadic task models, it can be successfully modeled using hyperbolic load-bound functions that are fitted to empirical performance measurements.*

## 1 Introduction

Device drivers are the software components for managing I/O devices. Traditionally, device drivers for common hardware devices (e.g.. network cards and hard disks) are implemented as part of the operating system kernel for performance reasons. Device drivers have also traditionally been a weak spot of most operating systems, especially in terms of accounting and control of the resources consumed by these software components. Each device driver's code may run in multiple (possibly concurrent) execution contexts which makes the resource accounting difficult, if not impossible. For instance, Linux device drivers are scheduled in a hierarchy of *ad hoc* mechanisms, namely hard interrupt service routines (ISR), *softirqs*, and process or thread contexts, in decreasing order of execution priorities.

While the traditional ways of scheduling device drivers can be tolerated in best-effort systems, they tend to present a problem for real-time systems. Real-time systems need to guarantee that certain workloads can be completed within specified time constraints. This implies that any workload within a real-time system must be amenable to *schedulability analysis*, which is defined as the application of abstract workload and scheduling models to predict the ability of the real-time system to meet all of its timeliness guarantees.

The workloads imposed by device drivers tend to be aperiodic, hard to characterize, and they defy schedulability analysis because much of their computational workload is often triggered by unpredictable events (e.g. arrival of network packets or completion of disk I/O). There may be blocking due to nonpreemptable critical sections within device drivers and preemption due to ISR code that executes in response to a hardware interrupt. The interference caused by device drivers on the execution of time-critical tasks, through

[*]Based upon work supported in part by the National Science Foundation under Grant No. 0509131, and a DURIP equipment grant from the Army Research Office.

such blocking and preemption, needs to be accurately modeled and included in the schedulability analysis of the system. In addition, the device drivers themselves may have response time constraints imposed by the need to maintain some quality of I/O services.

In this paper we present two approaches to *demand-based schedulability analysis* of systems including device drivers, based on a combination of analytically and empirically derived load-bound functions. Demand-based schedulability analysis views the schedulability analysis problem in terms of supply and demand. One defines a measure of computational demand and then shows that a system can meet all deadlines by proving that demand in any time interval cannot exceed the computational capacity of the available processors. This analysis technique has been successfully applied to several abstract workload models and scheduling algorithms, for both single and multiprocessor systems [9, 1, 3, 2].

Aperiodic device-driver tasks present a special challenge for demand-based schedulability analysis, because their potential computational demand is unknown. In principle, analysis would be possible if they were scheduled according to an algorithm that budgets compute time. However, the common practice in commodity operating systems is to schedule them using a combination of ad hoc mechanisms described above, for which it may be impractical or impossible to derive an analytical bound on the interference that the device driver tasks may cause other time-critical tasks. Two possible approaches for analysis:

1. Derive a load-bound function for the driver empirically.
2. Modify the way device driver tasks are scheduled in the operating system, to use a time-budgeting algorithm for which a load-bound function can be derived analytically.

In the rest of this paper we evaluate both of the above approaches, using a device driver as a case study – the Linux *e1000* driver for the Intel Pro/1000 family of Ethernet network interface adapters. We focus on *demand-based* schedulability analysis using *fixed-priority* scheduling in a *uniprocessor* environment.

## 2 Demand Analysis

Our view of demand analysis is derived from studies of traditional workload models [9, 1, 3, 2] which are based on the concepts of job and task. A *job* is a schedulable component of computational work with a release time (earliest start time), a deadline, and an execution time. The *computational demand* of a job $J$ in a given time interval $[a, b)$ for a given schedule, denoted by $demand_J(a, b)$, is defined to be the actual amount of processor time consumed by that job within the interval.

Suppose there is a single processor, scheduled according to a policy that is priority driven. Every job will be completed on time as long as the sum of its own execution time and the *interference* caused by the execution of other higher priority jobs within the same time window during which the job must be completed add up to no more than the length of the window. That is, suppose $\mathcal{J} = \{J_1, J_2, ...\}$ is the (possibly infinite) collection of jobs to be scheduled, numbered in order of decreasing priority. A job $J_k$ released at time $r_k$ with deadline $r_k + d_k$ and execution time $e_k$ will be completed by its deadline if

$$e_k + \sum_{i<k} demand_{J_i}(r_k, r_k + d_k) \leq d_k \qquad (1)$$

Traditional schedulability analysis relies on imposing constraints on the release times, execution times, and deadlines of the jobs of a system to ensure that inequality (1) is satisfied for every job. This is done by characterizing each job as belonging to one of a finite collection of *tasks*. A task is an abstraction for a collection of possible sequences of jobs.

The best understood type of task is *periodic*, with release times separated by a fixed period $p_i$, deadlines at a fixed offset $d_i$ relative to the release times, and actual execution times bounded by a fixed worst-case execution time $e_i$. A *sporadic* task is a slight relaxation of the periodic task model, in which the period $p_i$ is only a lower bound on the separation between the release times of the task's jobs.

The notions of computational demand and interference extend naturally to tasks. The function $demand^{\max}_{\tau_i}(\Delta)$ is the maximum of combined demands of all the jobs of $\tau_i$ in every time interval of length $\Delta$, taken over all possible job sequences of $\tau_i$. That is if $\mathcal{S}$ is the collection of all possible job sequences of $\tau_i$ then

$$demand^{\max}_{\tau_i}(\Delta) \overset{\text{def}}{=} \max_{S \in \mathcal{S}, t>0} \sum_{J \in S} demand_J(t - \Delta, t) \qquad (2)$$

Restated in terms of tasks, the same reasoning says that a task $\tau_k$ with relative deadline $d_k$ will always meet its deadline if the sum of its own execution time and the interference of higher priority tasks within any time window of length $d_k$ never exceeds $d_k$. That is, suppose there is a set of tasks $\tau = \tau_1, \ldots, \tau_n$, numbered in order of decreasing priority, and $d_k \leq p_k$. Then every job of

$\tau_k$ will complete within its deadline if

$$e_k + \sum_{i=1}^{k-1} demand^{\max}_{\tau_i}(d_k) \le d_k \qquad (3)$$

A core observation for preemptive fixed-priority scheduling of periodic and sporadic tasks is the following *traditional demand bound*:

$$demand^{\max}_{\tau_i}(\Delta) \le \left\lceil \frac{\Delta}{p_i} \right\rceil e_i \qquad (4)$$

This says that the maximum computation time of $\tau_i$ in any interval length $\Delta$ can be no more than the maximum execution time required by one job of $\tau_i$, multiplied by the maxumum number of jobs of $\tau_i$ that can execute in that interval. Replacing the maximum demand in (3) by the expression on the right in (4) leads to a well known response test for fixed-priority schedulability, *i.e.*, $\tau_k$ is always scheduled to complete by its deadline if

$$e_k + \sum_{i=1}^{k-1} \left\lceil \frac{d_k}{p_i} \right\rceil e_i \le d_k \qquad (5)$$

In Section 4 we report experiments in which we measured the actual interfering processor demand due to certain high priority tasks, over time intervals of different lengths. When we computed the maximum observed demand from that data, we observed that it never reached the level of the traditional demand bound, given by the expression on the right of (4). That is because the traditional demand *over-estimates* the actual worst-case execution time of $\tau_i$ in many intervals, by including the full execution time $e_i$ even in cases where the interval is not long enough to permit that. For example, suppose $p_i = d_i = 7$ and $e_i = 2$, and consider the case $\Delta = 8$. Since the release times of $\tau_i$ must be separated by at least 7, the maximum amount of time that $\tau_i$ can execute in any interval of length 8 is 3, not 4.

In this paper we introduce a *refined demand bound*, obtained by only including the portion of the last job's execution time that fits into the interval, as follows.

$$demand^{\max}_{\tau_i}(\Delta) \le je_i + \min(e_i, \Delta - jp_i) \qquad (6)$$

where

$$j \overset{\text{def}}{=} \left\lfloor \frac{\Delta}{p_i} \right\rfloor$$

The difference between the traditional demand bound and our refined bound in (6) is shown by Figure 1 for a periodic task with $p_i = 7$ and $e_i = 2$. The two bounds are equal between points that correspond to earliest and latest possible completion times of jobs, but the refined bound is tighter for other points.

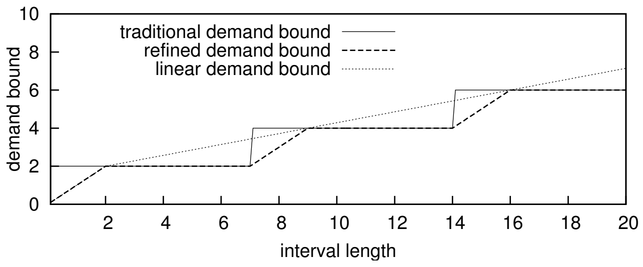

**Figure 1.** Comparison of the demand bounds of (5) and (6), for a periodic task with $p_i = 7$ and $e_i = 2$.

The diagonal line in the figure corresponds to a simplified upper bound function, obtained by interpolating linearly between the points $\Delta = jp_i + e_i$ at which the traditional and refined demand bounds converge. At these points, the expression on the right of (6) reduces to

$$(\frac{\Delta - e_i}{p_i} + 1)e_i = u_i(\Delta + p_i - e_i)$$

and so

$$demand^{\max}_{\tau_i}(\Delta) \le \min(\Delta, u_i(\Delta + p_i - e_i)) \qquad (7)$$

The above definitions and analyses can also be expressed in terms of the ratio of demand to interval length, which we call *load*. That is, $load_{\tau_i}(t - \Delta, t) \overset{\text{def}}{=} demand_{\tau_i}(t - \Delta, t)/\Delta$ and $load^{\max}_{\tau_i}(\Delta) \overset{\text{def}}{=} demand^{\max}_{\tau_i}(\Delta)/\Delta$. It follows from (3) that a task $\tau_k$ will always complete by its deadline if

$$\frac{e_k}{d_k} + \sum_{i=1}^{k-1} load^{\max}_{\tau_i}(d_k) \le 1 \qquad (8)$$

That is, to verify that a task $\tau_k$ always completes by its deadline, it is sufficient to add the percentage of CPU time used by all higher priority tasks and the percentage required for $\tau_k$ in any interval of length $d_k$. If this sum is less than or equal to one, then there is enough CPU time available for $\tau_k$ to finish its work on time.

The corresponding refined load-bound function of a periodic task can be derived by dividing (6) by $\Delta$, resulting in

$$load^{\max}_{\tau_i}(\Delta) \le \frac{je_i + \min(e_i, \Delta - jp_i)}{\Delta} \qquad (9)$$

where $j$ is defined as in (6), and a simplified *hyperbolic load bound* can be obtained by dividing (7) by $\Delta$, resulting in

$$load^{\max}_{\tau_i}(\Delta) \le \min(1, u_i(1 + \frac{p_i - e_i}{\Delta})) \qquad (10)$$

The refined load-bound function on the right of (9) and the hyperbolic approximation on the right of (10) are compared to the traditional load-bound function in Figure 2, for the same task as Figure 1.

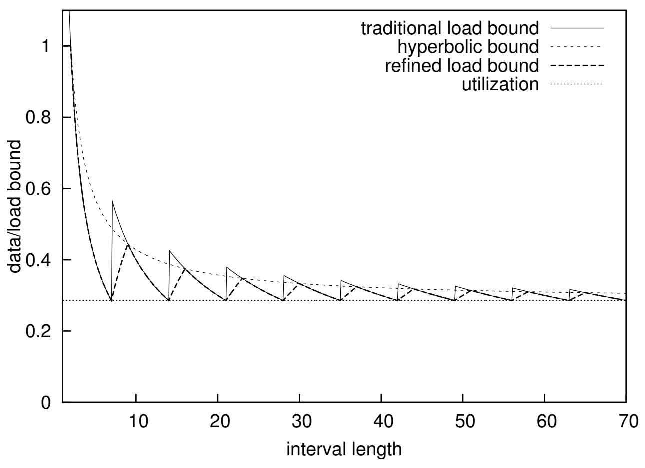

**Figure 2.** Load bounds for a periodic task with $p_i = 7$ and $e_i = d_i = 2$.

We find the load-based formulation more intuitive, since it allows us to view the interference that a task may cause other tasks as a percentage of the total available CPU time, which converges to the utilization factor $u_i = e_i/p_i$ for sufficiently long intervals. This can be seen in in Figure 2, where all the load bounds converge to the limit 2/7. Because of the critical zone property [9], an upper bound on the percentage interference a periodic or sporadic task would cause for any job of a given lower priority task can be discovered by reading the Y-value of any of these load-bound functions for the X-value that corresponds to the deadline of the lower priority task.

Demand-based schedulability analysis extends from periodic and sporadic tasks to non-periodic tasks through the introduction of *aperiodic server* thread scheduling algorithms, for which a demand-bound function similar to the one above can be shown to apply to even non-periodic tasks. The simplest such scheduling algorithm is the *polling server* [14], in which a task with a fixed priority level (possibly the highest) and a fixed execution budget is scheduled periodically and allowed to execute until it has consumed the budgeted amount of execution time, or until it suspends itself voluntarily (whichever occurs first). Other aperiodic server scheduling policies devised for use in a fixed-priority preemptive scheduling context include the Priority Exchange, Deferrable Server [16, 8], and Sporadic Server (not to be confused with sporadic task) algorithms [15, 10]. These algorithms improve upon the polling server by allowing a thread to suspend itself without giving up its remaining budget, and so are termed *bandwidth preserving algorithms*.

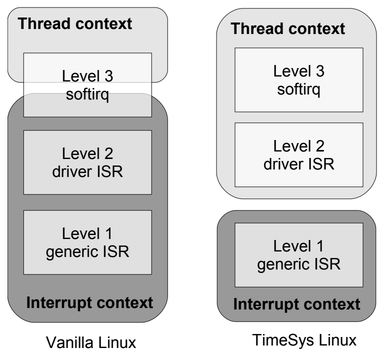

**Figure 3.** Comparison of execution contexts for vanilla and Timesys Linux.

## 3 The Linux e1000 Driver

Network interface device drivers are representative of the devices that present the biggest challenge for modeling and schedulability analysis, because they generate a very large workload with an unpredictable arrival pattern. Among network devices, we chose a gigabit Ethernet device for its high data rate, and the Intel Pro/1000 because it has one of the most advanced open-source drivers, namely the e1000 driver. This section describes how the e1000 driver is scheduled in the Linux kernel.

The Linux e1000 driver implements the new Linux API (NAPI) for network device drivers [11], which leaves the hardware interrupts for incoming packets disabled as long as there are queued received packets that have not been processed. The device interrupt is only re-enabled when the server thread has polled, discovered it has no more work, and so suspends itself. These mechanisms were originally developed to reduce receive live-lock but also has the effect of reducing the number of per-packet hardware interrupts.

The device-driven workload of the e1000 driver can be viewed as two device-driven tasks: (1) input processing, which includes dequeuing packets that the device has previously received and copied directly into system memory and the replenishing the list of DMA buffers available to the device for further input; (2) output processing, which includes dequeuing packets already sent and the enqueueing of more packets to send. In both cases, execution is triggered by a hardware interrupt, which causes execution of a hierarchy of handlers and threads.

The scheduling of the e1000 device-driven tasks can be described as occurring at three levels. The scheduling of the top two levels differs between the two Linux kernel versions considered here (Figure 3), which are the standard *"vanilla"* 2.6.16 kernel from *kernel.org*, and *Timesys Linux*, a version of the 2.6.16 kernel patched by Timesys Corporation to better support real-time applications.

**Level 1.** The hardware preempts the currently executing thread and transfers control to a generic interrupt service routine (ISR) which saves the processor state and eventually calls a Level 2 ISR installed by the device driver. The Level 1 processing is always preemptively scheduled at the device priority. The only way to control when such an ISR executes is to selectively enable and disable the interrupt at the hardware level.

**Level 2.** The driver's ISR does the minimum amount of work necessary, and then requests that the rest of the driver's work be scheduled to execute at Level 3 via the kernel's "softirq" (software interrupt) mechanism. In vanilla Linux this Level 2 processing is called directly from the Level 1 handler, and so it is effectively scheduled at Level 1. In contrast, Timesys Linux defers the Level 2 processing to a scheduled kernel thread, one thread per IRQ number on the x86 architectures.

**Level 3.** The softirq handler does the rest of the driver's work, including call-outs to perform protocol-independent and protocol-specific processing. In vanilla Linux, the Level 3 processing is scheduled via a complicated mechanism with two sub-levels: A limited number of softirq calls are executed ahead of the system scheduler, on exit from interrupt handlers, and at other system scheduling points. Repeated rounds of a list of pending softirq handlers are made, allowing each handler to execute to completion without preemption, until either all have been cleared or a maximum iteration count is reached. Any softirq's that remain pending are served by a kernel thread. The reason for this ad hoc approach is to achieve a balance between throughput and responsiveness. Using this mechanism produces very unpredictable scheduling results, since the actual instant and priority at which a softirq handler executes can be affected by any number of dynamic factors. In contrast, the Timesys kernel handles softirq's entirely in threads; there are two such threads for network devices, one for input processing and one for output processing.

The arrival processes of the e1000 input and output processing tasks generally need to be viewed as aperiodic, although there may be cases where the network traffic inherits periodic or sporadic characteristics from the tasks that generate it. The challenge is how to model the aperiodic workloads of these tasks in a way that supports schedulability analysis.

## 4 Empirical Load Bound

In this section we show how to model the workload of a device-driven task by an empirically derived load-bound function, which can then be used to estimate the preemptive interference effects of the device driver on the other tasks in a system.

For example, suppose one wants to estimate the total worst-case device-driven processor load of a network device driver, viewed as a single conceptual task $\tau_D$. The first step is to experimentally estimate $load^{\max}_{\tau_D}(\Delta)$ for enough values of $\Delta$ to be able to produce a plot similar to Figure 2 in Section 2. The value of $load^{\max}_{\tau_D}(\Delta)$ for each value of $\Delta$ is approximated by the maximum observed value of $demand_{\tau_D}(t-\Delta,t)/\Delta$ over a large number of intervals $[t-\Delta,t)$.

One way to measure the processor demand of a device-driven task in an interval is to modify the kernel, including the softirq and interrupt handlers, to keep track of every time interval during which the task executes. We started with this approach, but were concerned about the complexity and the additional overhead introduced by the fine-grained time accounting. Instead, we settled on the subtractive approach, in which the CPU demand of a device driver task is inferred by measuring the processor time that is left for other tasks.

To estimate the value of $demand_{\tau_D}(t-\Delta,t)$ for a network device driver we performed the following experiment, using two computers attached to a dedicated network switch. Host A sends messages to host C at a rate that maximizes the CPU time demand of C's network device driver. On system C, an application thread $\tau_2$ attempts to run continuously at lower priority than the device driver and monitors how much CPU time it accumulates within a chosen-length interval. All other activity on C is either shut down or run at a priority lower than $\tau_2$. If $\Delta$ is the length of the interval, and $\tau_2$ is able to execute for $x$ units of processor time in the interval, then the CPU demand attributed to the network device is $\Delta - x$ and the load is $(\Delta - x)/\Delta$.

It is important to note that this approach only measures CPU interference. It will not address memory cycle interference due to DMA operations. The reason is that most if not all of the code from $\tau_2$ will operate out of the processor's cache and therefore virtually no utilization of the memory bus will result from $\tau_2$. This effect, known as cycle stealing, can slow down a memory intensive task. Measurement of memory cycle interference is outside the scope of the present paper.

Each host had a Pentium D processor running in single-core mode at 3.0 GHz, with 2 GB memory and an Intel Pro/1000 gigabit Ethernet adapter, and was attached to a dedicated gigabit switch. Task $\tau_2$ was run using the SCHED_FIFO policy (strict preemptive priorities, with FIFO service among threads of equal priority) at a real-time priority just below that of the network softirq server threads. All its memory was locked into physical memory, so there were no other I/O activities (e.g. paging and swapping).

The task $\tau_2$ estimated its own running time using a technique similar to the Hourglass benchmark system [12]. It estimated the times of preemption events experienced by a thread by reading the system clock as frequently as possible and looking for larger jumps than would occur if the thread were to run between clock read operations without preemption. It then added up the lengths of all the time intervals where it was not preempted, plus the clock reading overhead for the intervals where it was preempted, to estimate amount of time that it was able to execute.

The first experiment was to determine the base-line preemptive interference experienced by $\tau_2$ when $\tau_D$ is idle, because no network traffic is directed at the system. That is, we measured the maximum processor load that $\tau_2$ can place on the system when no device driver execution is required, and subtracted the value from one. This provided a basis for determining the network device driver demand, by subtracting the idle-network interference from the total interference observed in later experiments when the network device driver was active.

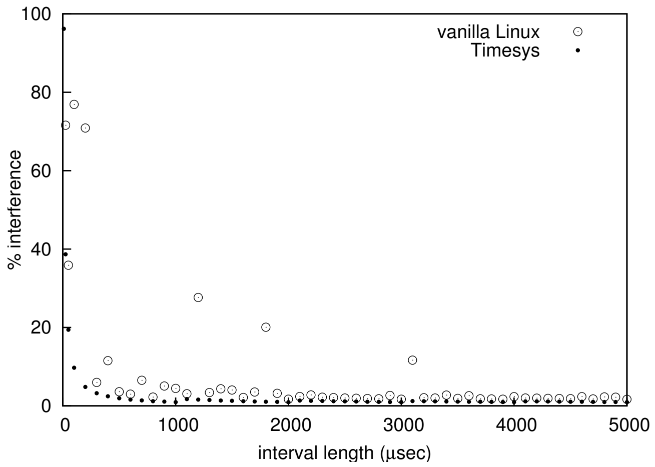

**Figure 4.** Observed interference with no network traffic.

Figure 4 shows the results of this experiment in terms of the percent interference observed by task $\tau_2$.

For this and the subsequent graphs, note that each data point represents the maximum observed preemptive interference over a series of trial intervals of a given length. This is a hard lower bound, and it is also a statistical estimate of the experimental system's worst-case interference over all intervals of the given length. Assuming the interference and the choice of trial intervals are independent, the larger the number of trial intervals examined the closer the observed maximum should converge to the system's worst-case interference.

The envelope of the data points should be approximately hyperbolic; that is, there should be an interval length below which the maximum interference is 100%, and there should be an average processor utilization to which the interference converges for long intervals. There can be two valid reasons for deviation from the hyperbola: (1) Periodic or nearly periodic demand, which results in a zig-zag shaped graph similar to line labeled "refined load bound" in Figure 2 (see Section 2); (2) not having sampled enough intervals to encounter the system's worst-case demand. The latter effects should diminish as more intervals are sampled, but the former should persist.

In the case of Figure 4 we believe that the tiny blips in the Timesys line around 1 and 2 msec are due to processing for the 1 msec timer interrupt. The data points for vanilla Linux exhibit a different pattern, aligning along what appear to be multiple hyperbolae. In particular, there is a set of high points that seems to form one hyperbola, a layer of low points that closely follows the Timesys plot, and perhaps a middle layer of points that seems to fall on a third hyperbola. This appearance is what one would expect if there were some rare events (or co-occurrences of events) that caused preemption for long blocks of time. When one of those occurs it logically should contribute to the maximum load for a range of interval lengths, up to the length of the corresponding block of preemption, but it only shows up in the one data point for the length of the trial interval where it was observed. The three levels of hyperbolae in the vanilla Linux graph suggest that there are some events or combinations of events that occur too rarely to show up in all the data points, but that if the experiment were continued long enough data points on the upper hyperbola would be found for all interval lengths.

Clearly the vanilla kernel is not as well behaved as Timesys. The high variability of data points for the vanilla kernel suggests that the true worst-case interference is much higher than the envelope suggested by the data. That is, if more trials were performed for each data point then higher levels of interference would be expected to occur throughout. By comparison, the observed maximum interference for Timesys appears to be bounded within a tight envelope over all interval lengths. The difference is attributed to Timesys' patches to increase preemptability.

The remaining experiments measured the behavior of the network device driver task $\tau_D$ under a heavy load, consisting of ICMP "ping" packets every 10 $\mu$sec. ICMP "ping" packets were chosen because they would execute entirely in the context of the device driver's receive thread, from actually receiving the packet through replying to it (TCP and UDP split execution between send and receive threads).

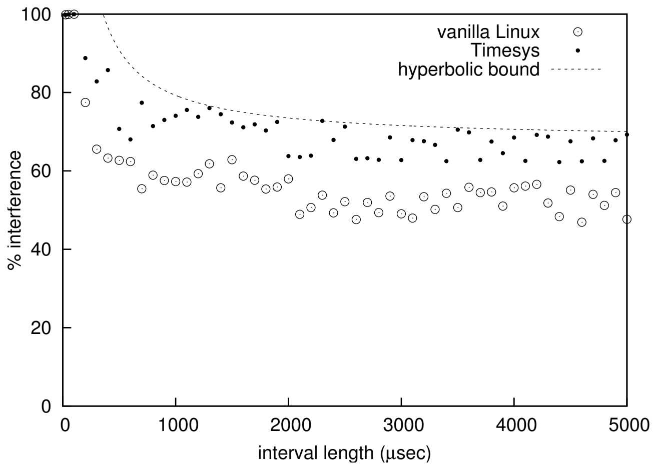

**Figure 5.** Observed interference with ping flooding, including reply.

Figure 5 shows the observed combined interference of the driver and base operating system under a network load of one ping every 10 $\mu$sec. The high variance of data points observed for the vanilla kernel appears to extend to Timesys. This indicates a rarely occurring event or combination of events that occurs in connection with network processing and causes a long block of preemption. We believe that this may be a "batching" effect arising from the NAPI policy, which alternates between polling and interrupt-triggered execution of the driver. A clear feature of the data is that the worst-case preemptive interference due to the network driver is higher with the Timesys kernel than the vanilla kernel. We believe that this is the result of additional time spent in scheduling and context-switching, because the network softirq handlers are executed in scheduled threads rather than borrowed context.

Given a set of data from experimental measurements of interference, we can fit the hyperbolic bound through application of inequality (9) from Section 2. There are several ways to choose the utilization and period so that the hyperbolic bound is tight. The method used here is: (1) eliminate any upward jogs from the data by replacing each data value by the maximum of the values to the right of it, resulting in a downward staircase function; (2) approximate the utilization by the value at the right most step; (3) choose the smallest period for which the resulting hyperbola intersects at least one of the data points and is above all the rest.

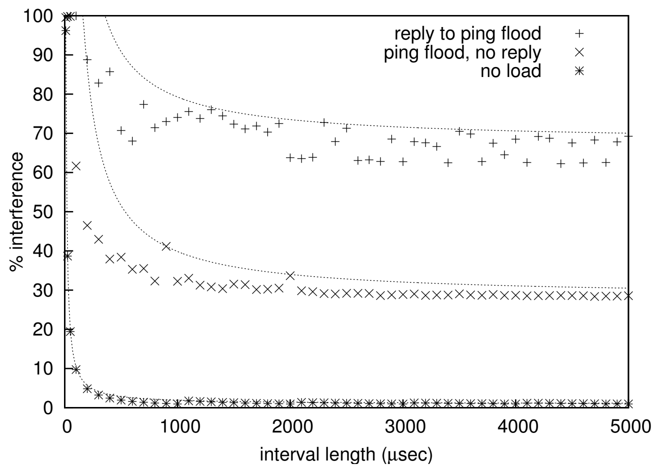

**Figure 6.** Observed interference with ping flooding, with no reply.

To carry the analysis further, an experiment was done to separate the load bound for receive processing from the load bound for transmit processing. The normal system action for a ping message is to send a reply message. The work of replying amounts to about half of the work of the network device driver tasks for ping messages. A more precise picture of the interference caused by just the network receiving task can be obtained by informing the kernel not to reply to ping requests. The graph in Figure 6 juxtaposes the observed interference due to the driver and base operating system with ping-reply processing, without ping-reply processing, and without any network load. The fitted hyperbolic load bound is also shown for each case. An interesting difference between the data for the "no reply" and the normal ping processing cases is the clear alignment of the "no reply" data into just two distinct hyperbolae, as compared to the more complex pattern for the normal case. The more complex pattern of variation in the data for the case with replies may be due to the summing of the interferences of these two threads, whose busy periods sometimes coincide. If this is true, it suggests a possible improvement in performance by forcing separation of the execution of these two threads.

Note that understanding these phenomena is not necessary to apply the techniques presented here. In fact the ability to model device driver interference without knowledge of the exact causes for the interference is the chief reason for using these techniques.

## 5 Interference *vs.* I/O Service Quality

This section describes further experiments, involving the device driver with two sources of packets and two hard-deadline periodic tasks. These were intended to explore how well empirical load bounds derived by the technique in Section 4 work with analytical load bounds for periodic tasks for whole-system schedulability analysis. We were also interested in comparing the degree to which scheduling techniques that reduce interference caused by the device-driver task for other tasks (e.g. lowering its priority or limiting its bandwidth through an aperiodic server scheduling algorithm); would affect the quality of network input service.

The experiments used three computers, referred to as hosts A, B, and C. Host A sent host C a heartbeat datagram once every 10 msec, host B sent a ping packet to host C every 10$\mu$sec (without waiting for a reply), and host C ran the following real-time tasks:

- $\tau_D$ is the device-driven task that is responsible for processing packets received and sent on the network interface (viewing the two kernel threads *softirq-net-rx* and *softirq-net-tx* as a single task).
- $\tau_1$ is a periodic task with a hard implicit deadline and execution time of 2 msec. It attempts one non-blocking input operation on a UDP datagram socket every 10 msec, expecting to receive a heartbeat packet, and counts the number of heartbeat packets it receives. The packet loss rate measures the quality of I/O service provided by the device driver task $\tau_D$.
- $\tau_2$ is another periodic task, with the same period and relative deadline as $\tau_1$. Its execution time was varied, and the number of deadline misses was counted at each CPU utilization level. The number of missed deadlines reflects the effects of interference caused by the device driver task $\tau_D$.

All the memory of these tasks was locked into physical memory, so there were no other activities. Their only competition for execution was from Level 1 and Level 2 ISRs. The priority of the system thread that executes the latter was set to the maximum real time priority, so that $\tau_D$ would always be queued to do work as soon as input arrived.

Tasks $\tau_1$ and $\tau_2$ were implemented by modifying the Hourglass benchmark [12], to accommodate task $\tau_1$'s nonblocking receive operations.

| Server | $\tau_1$ | $\tau_2$ | $\tau_D$ | OS |
|---|---|---|---|---|
| Traditional | high | med | hybrid | vanilla |
| Background | high | med | low | Timesys |
| Foreground | med | low | high | Timesys |
| Sporadic | high | low | med (SS) | Timesys |

**Table 1.** Configurations for experiments.

We tested the above task set in four scheduling configurations. The first was the vanilla Linux kernel. The other three used Timesys with some modifications of our own to add support for a Sporadic Server scheduling policy (SS). The SS policy was chosen because it is well known and is likely to be already implemented in the application thread scheduler of any real-time operating system, since it is the only aperiodic server scheduling policy included in the standard POSIX and Unix (TM) real-time API's.

The tasks were assigned relative priorities and scheduling policies as shown in Table 1. The scheduling policy was SCHED_FIFO except where SS is indicated.

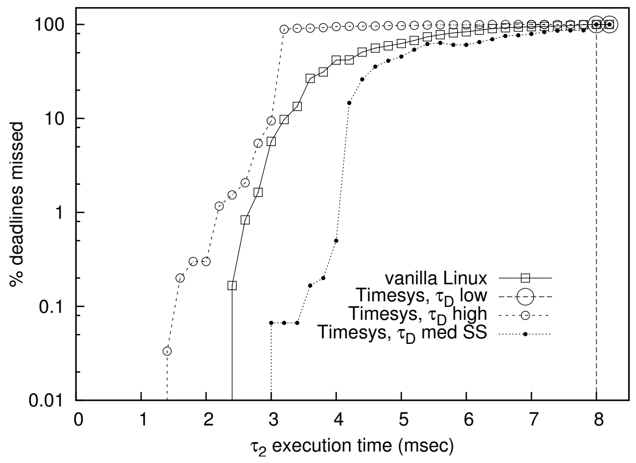

**Figure 7.** Percent missed deadlines of $\tau_2$ with interference from $\tau_1$ ($e_1 = 2$ and $p_1 = 10$) and $\tau_D$ subject to one PING message every 10 $\mu$sec.

Figures 7 and 8 show the percentage of deadlines that task $\tau_2$ missed and the number of heartbeat packets that $\tau_1$ missed, for each of the experimental configurations.

The Traditional Server experiments showed that the vanilla Linux two-level scheduling policy for softirq's causes $\tau_2$ to miss deadlines at lower utilization levels and causes a higher heartbeat packet loss rate for $\tau_1$ than the other driver scheduling methods. Nevertheless, the vanilla Linux behavior does exhibit some desirable properties. One is nearly constant packet loss rate, independent of the load from $\tau_1$ and $\tau_2$. That is due

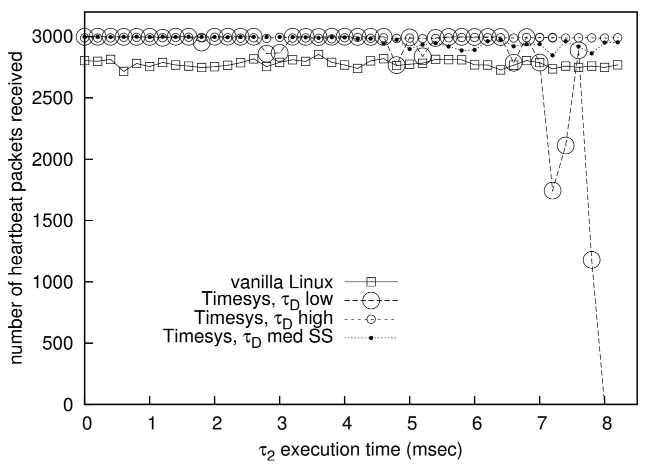

**Figure 8.** Number of heartbeat packets received by $\tau_1$ with interference from $\tau_2$ ($e_1 = 2$ and $p_1 = 10$) and $\tau_D$ subject to one PING message every 10 $\mu$sec.

to the ability of the driver to obtain some processing time at top priority, but only a limited amount. (See the description of Level 3 processing in Section 3 for details.) Another property, which is positive for soft-deadline applications, is that the missed deadline rate of $\tau_2$ degrades gracefully with increasing system load. These are also characteristics of an aperiodic scheduling algorithm, which the Linux policy approximates by allocating a limited rate of softirq handler executions at top priority and deferring the excess to be completed at low priority. However, the vanilla Linux policy is not simple and predictable enough to support schedulability analysis. Additionally, this strategy does not allow for user-level tuning of the device driver scheduling.

The Background Server experiments confirmed that assigning $\tau_D$ the lowest priority of the three tasks (the default for Timesys) succeeds in maximizing the probability of $\tau_2$ in meeting its deadlines, but it also gives the worst packet loss behavior. Figure 9 shows the combined load for $\tau_1$ and $\tau_2$. The values near the deadline (10) suggest that if there is no interference from $\tau_D$ or other system activity, $\tau_2$ should be able to complete within its deadline until $e_2$ exceeds 7 msec. This is consistent with the data in Figure 7. The heartbeat packet receipt rate for $\tau_1$ starts out better than vanilla Linux, but degenerates for longer $\tau_2$ execution times.

The Foreground Server experiments confirmed that assigning the highest priority to $\tau_D$ causes the worst deadline-miss performance for $\tau_2$, but also gives the best heartbeat packet receipt rate for $\tau_1$. The line labeled "$\tau_1 + \tau_D$" in Figure 10 shows the sum of the theoretical load bound for $\tau_1$ and the empirical hyperbolic load bound for $\tau_D$ derived in Section 4. By examining

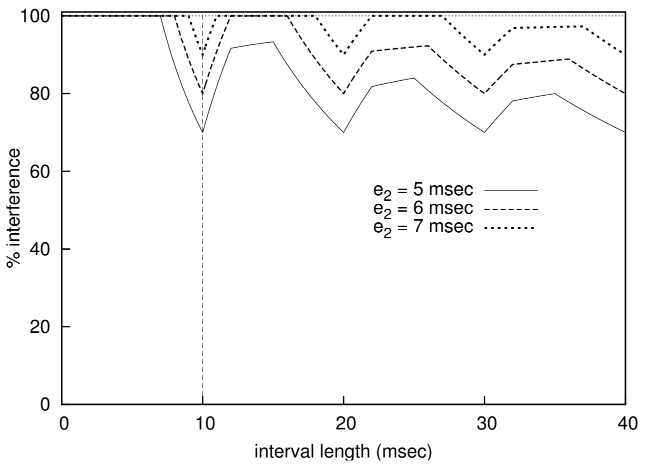

**Figure 9.** Sum of load-bound functions for $\tau_1$ and $\tau_2$, for three different values of the execution time $e_2$.

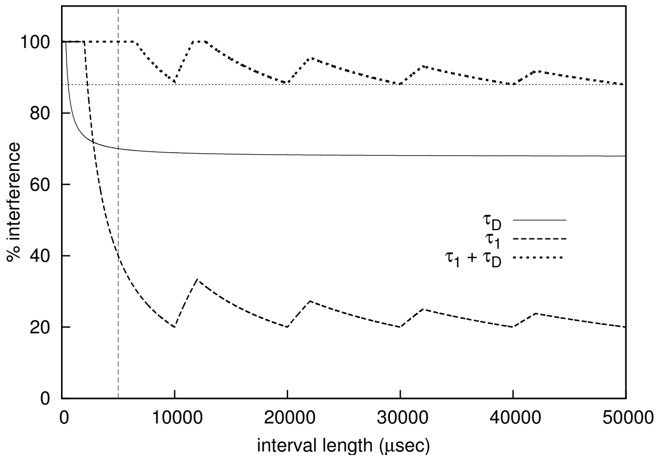

**Figure 10.** Individual load-bound functions for $\tau_1$ and $\tau_D$, and their sum.

the graph at the deadline (10000 $\mu$sec), and allowing some margin for release-time jitter, overhead and measurement error, one would predict that $\tau_2$ should not miss any deadlines until its execution time exceeds 1.2 msec. That appears to be consistent with the actual performance in Figure 7.

The Sporadic Server experiments represent an attempt to achieve a compromise that balances missed heartbeat packets for $\tau_1$ against missed deadlines for $\tau_2$, by scheduling $\tau_D$ according to a bandwidth-budgeted aperiodic server scheduling algorithm, running at a priority between $\tau_1$ and $\tau_2$. This has the effect of reserving a fixed amount of high priority execution time for $\tau_D$, effectively lowering the load bound curves. This allows it to preempt $\tau_2$ for the duration of the budget, but later reduces its priority to permit $\tau_2$ to execute, thereby increasing the number of deadlines $\tau_2$ is able to meet. The Sporadic Server algorithm implemented here uses the native (and rather coarse) time accounting granularity of Linux, which is 1 msec. The server budget is 1 msec; the replenishment period is 10 msec; and the number of outstanding replenishments is limited to two. It can be seen in figure 7 that running the experiments on the SS implementation produces data that closely resembles the behavior of the vanilla Linux kernel. (This is consistent with our observations on the similarity of these two algorithms in the comments on the Traditional Server experiments above.) Under ideal circumstances the SS implementation should not allow $\tau_2$ to miss a deadline until its execution time exceeds the sum of its own initial budget and the execution time of $\tau_1$. In this experiment our implementation of the SS fell short of this by 3 msec. In continuing research, we plan to narrow this gap by reducing the accounting granularity of our implementation and increasing the number of pending replenishments, and determine how much of the currently observed gap is due to the inevitable overhead for time accounting, context switches, and priority queue reordering.

## 6 Related Work

Previous research has considered a variety of techniques for dealing with interference between interrupt-driven execution of device-driver code and the scheduling of application threads. We classify these techniques into two broad groups, according to whether they apply before or after the interrupt occurs.

The first technique is to "schedule" hardware interrupts in a way that reduces interference, by reducing the number of interrupts, or makes it more predictable, by limiting when they can occur. On some hardware platforms, including the Motorola 68xxx series of microprocessors, this can be done by assigning different hardware priorities to different interrupts. The most basic approach to scheduling interrupts involves enabling and disabling interrupts intelligently. The Linux network device driver model called NAPI applies this concept to reduce hardware interrupts during periods of high network activity[11]. Regehr and Duongsaa [13] propose two other techniques for reducing interrupt overloads, one through special hardware support and the other in software. RTLinux can be viewed as also using this technique. That is, to reduce interrupts on the host operating system RTLinux interposes itself between the hardware and the host operating system[18]. In this way it relegates all device driver execution for the host to background priority, unless there is a need for better I/O performance. In the latter case, RTLinux allows device driver code to run as a RTLinux thread (see below).

The second technique, followed in the current paper, is to defer most interrupt-triggered work to scheduled threads. Hardware interrupt handlers are kept as short and simple as possible. They only serve to notify a scheduler that it should schedule the later execution of a thread to perform the rest of the interrupt-triggered work. There are variations to this approach, depending on whether the logical interrupt handler threads execute in borrowed (interrupt) context or in independent contexts (e.g. normal application threads), and on whether they have an independent lower-level scheduler (e.g. RTLinux threads or vanilla Linux softirq handlers) or are scheduled via the same scheduler as normal application threads. The more general the thread scheduling mechanism, the more flexibility the system developer has in assigning an appropriate scheduling policy and priority to the device-driven threads. The job of bounding device driver interference then focuses on analyzing the workload and scheduling of these threads. This technique has been the subject of several studies, including [7, 4, 17, 5], and is implemented in Windows CE and real-time versions of the Linux kernel.

Facchinetti et al. [6] recently proposed an instance of the work deferral approach, in which a system executes all driver code as one logical thread, at the highest system priority. The interrupt server has a CPU time budget, which imposes a bound on interference from the ISRs. They execute the ISRs in a non-preemptable manner, in interrupt context, ahead of the application thread scheduler. Their approach is similar to the softirq mechanism of the vanilla Linux system, in that both schedule interrupt handlers to run at the highest system priority, both execute in interrupt context, and both have a mechanism that limits server bandwidth consumption. However, time budgets are enforced directly in [6].

Zhang and West [19] recently proposed another variation of the work deferral approach, that attempts to minimize the priority of the bottom halves of driver code across all current I/O consuming processes. The algorithm predicts the priority of the process that is waiting on some queued I/O, and then executes the bottom half in its own thread at the highest predicted priority per interrupt. Then it charges the execution time to the predicted process. This approach makes sense for device driver execution that can logically be charged to an application process.

The above two techniques partially address the problem considered in this paper. That is, they restructure the device-driven workload in ways that potentially allow more of it to be executed below interrupt priority, and schedule the execution according to a policy that can be analyzed if the workload can be modeled. However, they do not address the problem of how to model the workload that has been moved out of the ISRs, or how to model the workload that remains in the ISRs.

A difference between the Facchinetti approach and our use of aperiodic server scheduling is that we have multiple threads, at different priorities, executing in independent contexts and scheduled according to standard thread scheduling policies which are also available to application threads. We have observed that different devices (i.e. NIC, disk controller, etc) all generate unique workloads, which we believe warrant different scheduling strategies and different time budgets. In contrast, all devices in the Facchinetti system are forced to share the same budget and share the same priority; the system is not able to distinguish between different priority levels of I/O, and is forced to handle all I/O in FIFO order. Imagine a scenario where the real time system is flooded with packets. In the Facchinetti system the NIC could exhaust the ISR server's budget. If a high priority task requests disk I/O while the ISR server's budget is exhausted, the disk I/O will be delayed until the ISR server budget is replenished, and the high priority task may not receive its disk service in time to meet its deadline. This scenario is pessimistic, but explains our motivation to move ISR execution into multiple fully-schedulable threads.

A difference between the Zhang and West approach and ours is that we focus on the case where there is no application process to which the device-driven activity can logically be charged. Our experiments use ICMP packets, which are typically processed in the context of the kernel and cannot logically be charged to a process.

Another difference is that our model is not subject to a middle-priority process delaying the execution of a higher priority process, by causing a backlog in the bottom-half processing of I/O for a device on which the high priority process depends. Consider a system with three real time processes, at three different priorities. Suppose the low priority process initiates a request for a stream of data over the network device, and that between packets received by the low priority process, the middle-priority process (which does not use the network device) wakes up and begins executing. Under the Zhang and West scheme, the network device server thread would have too low priority for the network device's bottom half to preempt the middle-priority process, and so a backlog of received packets would build up in the DMA buffers. Next, suppose the high priority process wakes up and during its execution, attempts to read from the network device. This will raise the bottom half's priority to that of the high priority process. However, since the typical network device driver handles packets in FIFO order, the bottom half is forced to work through the backlog of the low-priority process's input before it gets to the packet destined for the high priority process. This additional delay could be enough to cause the high priority process to miss its deadline. That would not have happened if the low-priority packets had been cleared out earlier, as if the device bottom half had been able to preempt the middle-priority task. In contrast, with our approach the bottom half still handles incoming packets in FIFO order, but by executing the bottom half in a server with a budget of high priority time we are able to empty the incoming DMA queue more frequently. This can prevent the scenario above from occurring unless the input rate exceeds the bottom-half server's budgeted bandwidth.

## 7 Conclusion

We have described two ways to approach the problem of accounting for the preemptive interference effects of device driver tasks in demand-based schedulability analysis. One is to model the worst-case interference of the device driver by a hyperbolic load-bound function derived from empirical performance data. The other approach is to schedule the device driver by an aperiodic server algorithm that budgets processor time consistent with the analytically derived load-bound function of a periodic task. We experimented with the application of both techniques to the Linux device driver for Intel Pro/1000 Ethernet adapters.

The experimental data show hyperbolic load bounds can be derived for base system activity, network receive processing, and network transmit processing. Further, the hyperbolic load bounds may be combined with analytically derived load bounds to predict the schedulability of hard-deadline periodic or sporadic tasks. We believe this technique of using empirically derived hyperbolic load-bound functions to model processor interference may also have potential applications outside of device drivers, to aperiodic application tasks that are too complex to apply any other load modeling technique.

The data also show preliminary indications that aperiodic-server scheduling algorithms, such as Sporadic Server, can be useful in balancing device driver interference and quality of I/O service. This provides an alternative in situations where neither of the two extremes otherwise available will do, *i.e.*, where running the device driver at a fixed high priority causes unacceptable levels of interference with other tasks, and running the device driver at a fixed lower priority causes unacceptably low levels of I/O performance.

In future work, we plan to study other device types, and other types of aperiodic server scheduling algorithms. We also plan to extend our study of empirically derived interference bounds to include memory cycle interference. As mentioned in this paper, our load measuring task can execute out of cache, and so does not experience the effects of memory cycle stealing due to DMA. Even where there is no CPU interference, DMA memory cycle interference may increase the time to complete a task past the anticipated worst-case execution time, resulting in missed deadlines. We plan to perform an analysis of DMA interference on intensive tasks. By precisely modeling these effects, increases in the execution time due to cycle stealing will be known and worst-case execution times will be more accurately predicted. Further, by coordinating the DMA and memory intensive tasks, the contention for accessing memory can be minimized.

## References

[1] N. C. Audsley, A. Burns, M. Richardson, and A. J. Wellings. Hard real-time scheduling: the deadline monotonic approach. In *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, Atlanta, GA, USA, 1991.

[2] T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In I. Lee, J. Y.-T. Leung, and S. Son, editors, *Handbook of Real-time and Embedded Systems*. CRC Press, 2007. (to appear).

[3] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. *Proc. 11th IEE Real-Time Systems Symposium*, pages 182–190, 1990.

[4] L. L. del Foyo, P. Meja-Alvarez, and D. de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 14–23, San Jose, CA, Apr. 2006.

[5] P. Druschel and G. Banga. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proc. 2nd USENIX symposium on operating systems design and implementation*, pages 261–275, Oct. 1996.

[6] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *Proc. 17th IEEE Euromicro Conference on Real-Time Systems*, Palma de Mallorca, July 2005.

[7] S. Kleiman and J. Eykholt. Interrupts as threads. *ACM SIGOPS Operating Systems Review*, 29(2):21–26, Apr. 1995.

[8] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proc. 8th IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

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

[10] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

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

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

[13] J. Regehr and U. Duongsaa. Preventing interrupt overload. In *Proc. 2006 ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems*, pages 50–58, Chicago, Illinois, June 2005.

[14] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritizing preemptive scheduling. In *Proc. 7th IEEE Real-Time Sytems Symposium*, 1986.

[15] B. Sprunt, L. Sha, and L. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[16] J. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in real-time environments. *IEEE Trans. Computers*, 44(1):73–91, Jan. 1995.

[17] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. *IEEE Trans. Networking*, 1(5):554–565, Oct. 1993.

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

[19] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *Proc. 27th Real Time Systems Symposium*, Rio de Janeiro, Brazil, Dec. 2006.

## Acknowledgment

The authors are grateful to Timesys Corporation for providing access to their distribution of the Linux kernel at a reduced price. We are also thankful to the anonymous members of the RTAS 2007 program committee for their suggestions.