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

M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan†, A. 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 commodity operating systems that control I/O devices. The computation 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, using a Linux network device driver as a case study. 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 the Intel Pro/1000 gigabit Ethernet device under Linux. Our results show that, even though the network device driver defies conventional demand analysis by not conforming to either periodic or sporadic task models, it can be successfully modeled using hyperbolic load-bound functions that are fitted to empirical performance measurements.

*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.

†Dr. Gopalan recently moved to the University of Binghamton, New York.

1 Introduction

Device drivers are the software components responsible for managing I/O devices of any system. Traditionally, device drivers for common hardware devices, such as network cards and hard disks, are implemented as part of the operating system kernel in order to provide them with the execution privileges required to access these devices. 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 execution contexts (possibly concurrent) 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 – and might often aid in optimizing performance – 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 in nature and are hard to characterize, and they defy clean schedulability analysis, primarily because much of their computational workload is often triggered by unpredictable events, such as arrival of network packets or completion of disk I/O operations. There may be blocking due to nonpreemptable criti-

cal 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 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 analytically derive a bound on the interference that the device driver tasks may cause other time-critical tasks. So, there are two possible approaches left 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 an 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 specific device driver as a case study – the Linux *e1000* driver for the Intel Pro/1000 family of Ethernet network interface adapters. We focus specifically 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 workloads 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 $[a, b)$, denoted by $demand_J(a, b)$, is defined to be the amount of processor time consumed by that job within the interval.

Suppose there is a single processor, scheduled according to a policy that is work conserving (meaning the processor is never idle while there is work ready to be executed), and $\mathcal{J}$ is the collection (possibly infinite) of jobs to be scheduled. It follows that if a job J is released at time r_J with deadline $r_J + d_J$ and execution time e_J then it can be completed by its deadline if

$$e_J + \sum_{J' \in \mathcal{J}, J' \neq J} demand_{J'}(r_J, r_J + d_J) \leq d_J \qquad (1)$$

That is, 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 jobs within the same time window during which the job must be completed add up to no more than the length of the window.

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*. A task τ_i is periodic if its jobs have 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 extended naturally to tasks. The function $demand_{\tau_i}^{\max}(\Delta)$ is the maximum of combined demands of all the jobs of τ_i in every time interval of length Δ, taken over all possible job sequences of τ_i. That is if $\mathcal{S}$ is the collection of all possible job sequences of τ_i then

$$demand_{\tau_i}^{\max}(\Delta) \stackrel{\text{def}}{=} \max_{S \in \mathcal{S}, t > 0} \sum_{J \in S} demand_J(t - \Delta, t) \quad (2)$$

It follows from (1) that if $d_i \leq p_i$ then each job of τ_k

will complete by its deadline as long as

$$e_k + \sum_{i \neq k} demand^{\max}_{\tau_i}(d_k) \leq d_k \quad (3)$$

A core observation for preemptive fixed-priority scheduling of periodic and sporadic tasks is that

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

Replacing the maximum demand in (3) by the expression on the right in (4) leads to the well known response time test for fixed-priority schedulability, *i.e.*, τ_k is always scheduled to complete by its deadline if

$$e_k + \sum_{i \neq k} \left\lceil \frac{d_k}{p_i} \right\rceil e_i \leq d_k \quad (5)$$

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

$$\frac{e_k}{d_k} + \sum_{i \neq k} load^{\max}_{\tau_i}(d_k) \leq 1 \quad (6)$$

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 the solid-line labeled "traditional load bound" in Figure 1, for a periodic task with $p_i = d_i = 7$ and $e_i = 2$, which converges clearly to the limit 2/7. Because of the critical zone property [9], an upper bound on the percentage interference this task would cause for any job of a given lower priority task can be discovered by reading the Y-value for the X-value that corresponds to the deadline of the lower priority task. The other functions in Figure 1 are a refined load bound and hyperbolic approximation which are explained in Section 4.

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], that is, a periodic task with a fixed priority level (possibly the highest) and a fixed execution capacity. A polling server is scheduled periodically at a given (high) priority, and allowed to execute at that priority until it has consumed up to a fixed budgeted amount of execution time, or until it suspends itself voluntarily (whichever occurs first). Other aperiodic server scheduling policies devised for use in a a fixed-priority preemptive scheduling context include the Priority Exchange, Deferrable Server [16, 7], 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*.

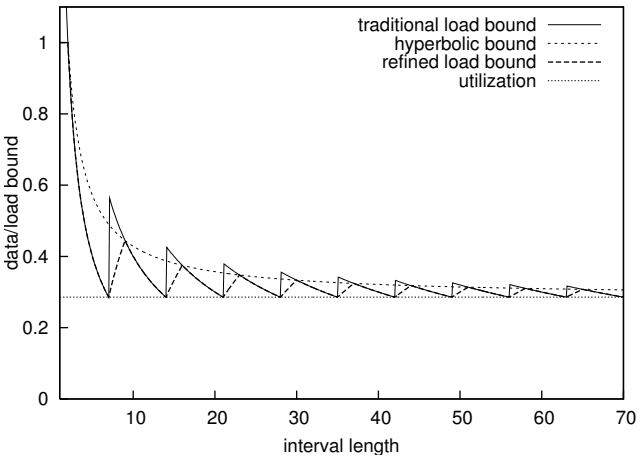

Figure 1. Load bounds for a periodic task with $p_i = 7$ and $e_i = d_i = 2$.

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 was originally developed to reduce receive live-lock but also has the effect of reducing the number of per-packet hardware interrupts. NAPI 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.

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 enqueue-ing 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, 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 device 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 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. 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 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 τ_D. The first step is to experimentally estimate $load_{\tau_D}^{\max}(\Delta)$ for enough values of Δ to be able to produce a plot similar to Figure 1 in Section 2. The value of $load_{\tau_D}^{\max}(\Delta)$ for each value of Δ 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 described below, 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 τ_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 τ_2. If Δ is the length of the interval, and τ_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 $(\Delta - x)/\Delta$.

The results reported here for the e1000 driver were obtained in this fashion. Both hosts 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 were attached to a dedicated gigabit switch. Task τ_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 was no page swapping activity in the system, and there were no other I/O activities.

The task τ_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 is was able to execute.

The first experiment was to determine the base-line preemptive interference experienced by τ_2 when τ_D is idle, because no network traffic is directed at the system. That is, we measured the maximum processor load that τ_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.

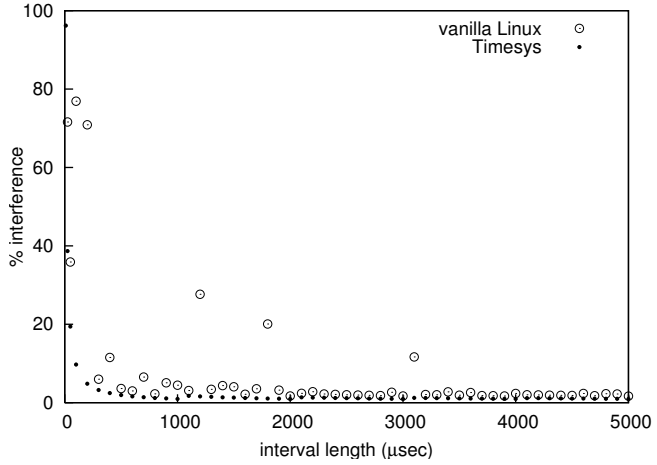

Figure 2. Observed interference with no network traffic.

Figure 2 shows the results of this experiment in terms of the percent interference observed by task τ_2.

When interpreting this and the subsequent graphs below, one should keep in mind 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 true 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 true worst-case interference.

The shape of the envelope of the data points should always 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 ideal hyperbola: (1) Periodic or nearly periodic demand, which results in a zig-zag shaped graph similar to line labeled "refined load bound" in Figure 1 (see Section 2); (2) not having sampled enough intervals to encounter the worst-case demand. The latter effects should diminish as more intervals are sampled, but the former should persist.

In the case of Figure 2 we believe that the tiny blips in the Timesys line around 1 msec 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 forms 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's patches to increase preemptability.

The remaining experiments measured the behavior of the network device driver task τ_D under a heavy load, consisting of ICMP "ping" packets every 10 μsec.

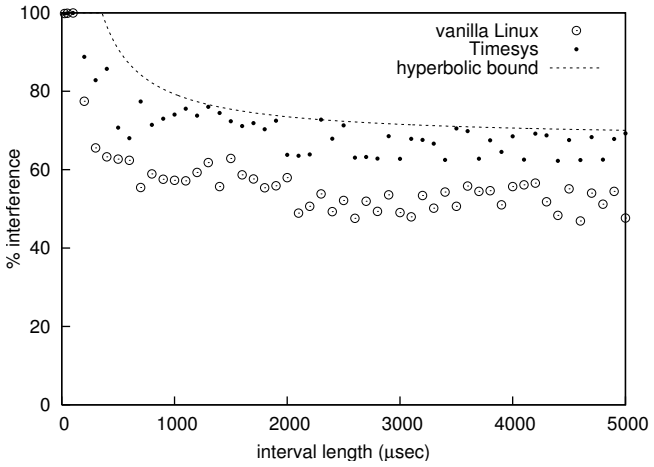

Figure 3. Observed interference with ping flooding, including reply.

Figure 3 shows the observed combined interference of the driver and base operating system interference under a network load of one ping every 10 μsec. The high variance of data points observed for the vanilla kernel in Figure 2 appears to now extend to the Timesys kernel. 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 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.

The line labeled "hyperbolic bound" in Figure 3 is a hyperbolic load-bound function, derived by fitting the experimental data to the formula of a load-bound function for periodic tasks. The derivation of the hyperbolic bound starts with the demand- and load-bound functions for periodic tasks in Section 2. First observe that the ceiling function in the traditional demand bound $\lceil \Delta/p_i \rceil e_i$ effectively includes the *entire* execution time of the last job of τ_i that is released in the interval, even if that is longer than the remaining time in the interval. If one were to measure the actual execution times of τ_i over a sufficiently large number of schedules, and compute the maximum demand from that data, as we did in the experiment reported above, there are interval lengths for which this bound could never actually be achieved. A tighter demand bound can be obtained by only including the portion of the last job's execution time that fits into the interval, as follows.

$$demand_{\tau_i}^{\max}(\Delta) \leq j_{\tau_i,\Delta} e_i + \min(e_i, \Delta - j_{\tau_i,\Delta} p_i) \quad (7)$$

where

$$j_{\tau_i,x\Delta} \stackrel{\text{def}}{=} \left\lfloor \frac{d_i}{p_i} \right\rfloor$$

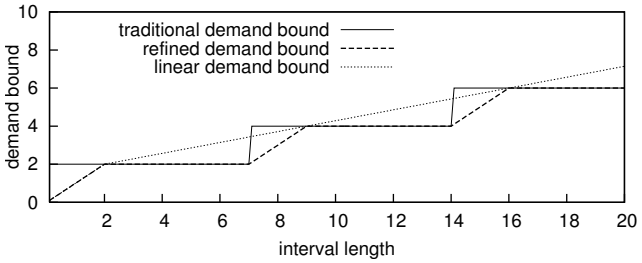

Figure 4. Comparison of the demand bounds of (5) and (7), for a periodic task with $p_i = 7$ and $e_i = 2$.

The traditional demand bound in (5) of Section 2 and the refined bound in (7) above are juxtaposed in Figure 4 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. The figure also shows how the refined bound leads to a linear demand bound function that is tighter than the traditional demand bound for some interval lengths.

The corresponding refined load-bound function and its hyperbolic approximation can be seen in Figure 1. A formula for the hyperbolic load bound of a periodic task with utilization $u_i = e_i/p_i$ and period p_i is

$$load_{\tau_i}^{\max}(\Delta) \leq \min(1, u_i(1 + \frac{p_i(1-u_i)}{\Delta})) \quad (8)$$

Given a set of data from experimental measurements of interference there are several reasonable ways to choose the utilization and period so that the hyperbolic bound is tight for the data. The method used here is: (1) eliminate any downward jobs from the data, by replacing each data value by the maximum of the values

to the left 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.

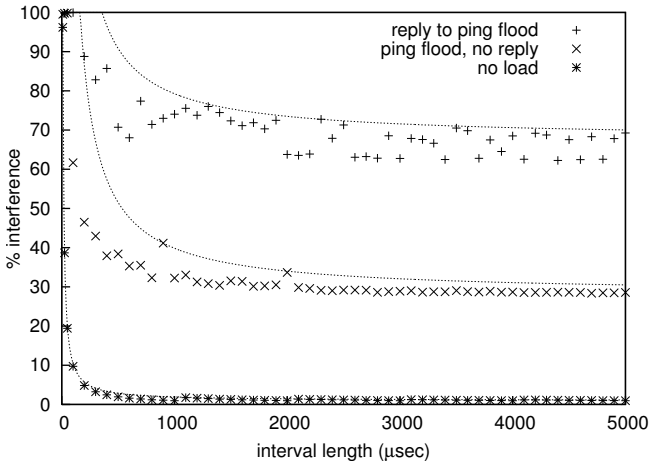

Figure 5. Observed interference with ping flooding, with no reply.

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 reply to ping requests. The graph in Figure 5 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 detailed 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 a device driver load bound derived empirically by the technique described in Section 4 works in combination with analytically derived 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, such as 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 10μsec (without waiting for a reply), and host C ran the following real-time tasks:

- τ_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).
- τ_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 τ_D.
- τ_2 is another periodic task, with the same period and relative deadline as τ_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 τ_D.

All the memory of these tasks was locked into physical memory, so there was no page swapping activity. 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 τ_D would always be queued to do work as soon as input arrived.

Tasks τ_1 and τ_2 were implemented by modifying the Hourglass benchmark [12], to accommodate task τ_1's nonblocking receive operations.

We tested the above task set in four scheduling configurations. The first was the vanilla Linux kernel. The

Server	τ_1	τ_2	τ_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.

other three used Timesys with some modifications of our own to add support for a Sporadic Server scheduling policy (SS). 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.

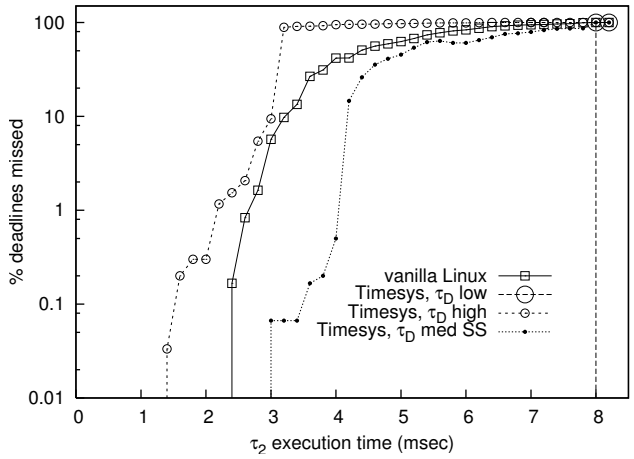

Figure 6. Percent missed deadlines of τ_2 with interference from τ_1 ($e_1 = 2$ and $p_1 = 10$) and τ_D subject to one PING message every 10 μsec.

Figures 6 and 7 show the percentage of deadlines that task τ_2 missed and the number of heartbeat packets that τ_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 τ_2 to miss deadlines at lower utilization levels and causes a higher heartbeat packet loss rate for τ_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 τ_1 and τ_2. That is due 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 τ_2 degrades gracefully with increasing system load. These are two characteristics of an aperiodic schedul-

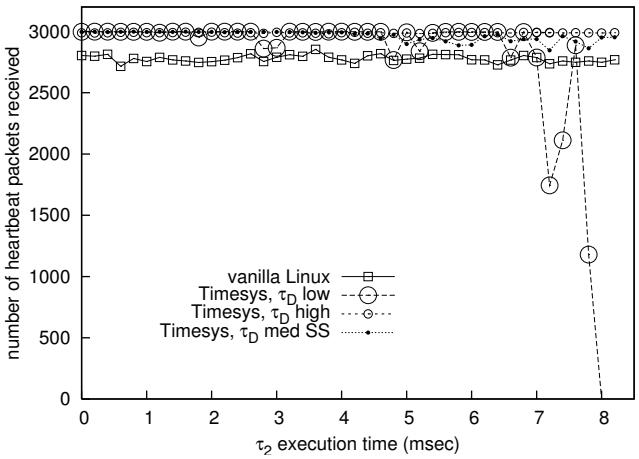

Figure 7. Number of heartbeat packets received by τ_1 with interference from τ_2 ($e_1 = 2$ and $p_1 = 10$) and τ_D subject to one PING message every 10 μsec.

ing algorithm like Sporadic Server, 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 τ_D the lowest priority of the three tasks (the default for Timesys) succeeds in maximizing the success of τ_2 in meeting its deadlines, it but it also gives the worst packet loss behavior. Figure 8 shows the combined load τ_1 and τ_2. The values near the deadline (10) suggest that if there is no interference from τ_D or other system activity, τ_2 should be able to complete within its deadline until e_2 exceeds 7 msec. This is consistent with the data in Figure 6. The heartbeat packet receipt rate for τ_1 starts out better than vanilla Linux, but degenerates for longer τ_2 execution times.

The Foreground Server experiments confirmed that assigning the highest priority to τ_D causes the worst deadline-miss performance for τ_2, but also gives the best heartbeat packet receipt rate for τ_1. The line labeled "$\tau_1 + \tau_D$" in Figure 9 shows the sum of the theoretical load bound for τ_1 and the empirical hyperbolic load bound for τ_D derived in Section 4. By examining the graph at the deadline (10000 μsec), and allowing some margin for release-time jitter, overhead and measurement error, one would predict that τ_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 6.

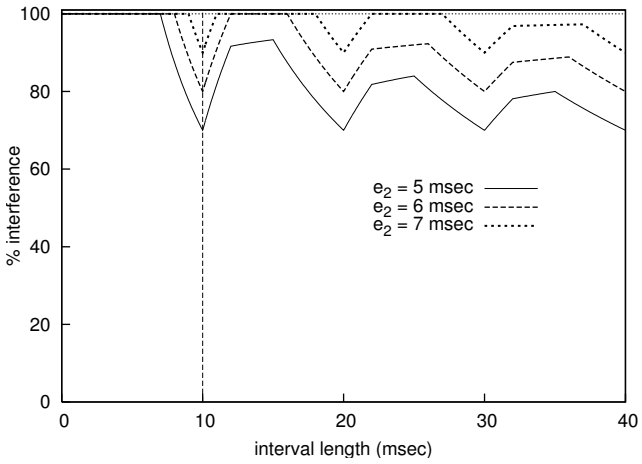

Figure 8. Sum of load-bound functions for τ_1 and τ_2, for three different values of the execution time e_2.

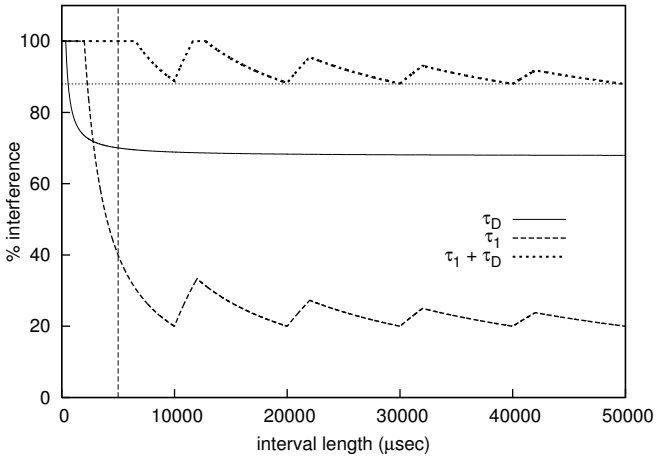

Figure 9. Individual load-bound functions for τ_1 and τ_D, and their sum.

The Sporadic Server experiments represent an attempt to achieve a compromise that balances missed heartbeat packets for τ_1 against missed deadlines for τ_2, by scheduling τ_D according to the Sporadic Server (SS) scheduling algorithm, running at a priority between τ_1 and τ_2. This has the effect of reserving a fixed amount of high priority execution time for τ_D. This allows it to preempt τ_2 for the duration of the budget, but later reduces its priority to permit τ_2 to execute, thereby increasing the number of deadlines τ_2 is able to meet. The version of 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 6 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 τ_2 to miss a deadline until its execution time exceeds the sum of its own initial budget and the execution time of τ_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

Several ideas have been proposed for reducing the priority level and duration of device driver interference, through controlling hardware interrupts and moving device driver work from ISRs to normally scheduled threads. One idea is to require that devices drivers satisfy certain structural constraints, as in TinyOS [8]. This approach is promising for new drivers, but has not yet been explored thoroughly with respect to more complex operating systems and devices. Another technique is to keep hardware interrupt handlers short and simple, and perform the bulk of the interrupt-triggered work of device drivers in normally scheduled kernel threads [6, 4, 17, 5]. The job of bounding device driver interference then focuses on analyzing the workload and scheduling of these threads. This practice is implemented in Windows CE and real-time versions of the Linux kernel. Other ideas have been proposed for reducing preemptive effects of device drivers by 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 takes interrupt reduction to an extreme, interposing itself between all hardware interrupts and the host operating system [18]. This relegates all device driver execution to background priority, until one discovers a need for better I/O performance and allows device driver code to run as a RTLinux thread. These techniques are complementary to 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 at lower priority, but do not address the problem of how to model the remaining ISR workload, or how to model the workload that has been moved to scheduled threads.

7 Conclusion

We have proposed 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 the Sporadic Server algorithm, which budgets processor time consistent with the analytically derived load-bound function of a periodic task.

We applied both techniques to the Linux device driver for Intel Pro/1000 Ethernet adapters, and provided data from experiments with this driver. The data show hyperbolic load bounds can be derived for base system activity, network receive processing, and network transmit processing, and may be combined with analytically derived load bounds to predict the schedulablility of hard-deadline periodic or sporadic tasks.

We believe the technique of using empirically derived hyperbolic load-bound functions has potential applications outside of device drivers, to aperiodic application tasks that are too complex to apply any other load modeling technique.

The experiments have also provided preliminary indications that the aperiodic-server scheduling algorithms, including Sporadic Server, can be applied to achieve a balance between device driver interference and quality of I/O service. This provides an alternative to the two extremes otherwise available, *i.e.*, to run the device driver at a fixed high priority and suffer unacceptable levels of interference with other tasks, or to run the device driver at a fixed lower priority and suffer unacceptably low levels of I/O performance.

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. *Journal of Embedded Computing*, 2006. (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 *OSDI '96 Proc. 2nd USENIX symposium on operating systems design and implementation*, pages 261–275, Oct. 1996.

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

[7] 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.

[8] P. Levis, D. G. S. Madden, J. Polastre, R. Szewzyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *Proc. 1st USENIX/ACM symposium on network systems design and implementation (NSDI)*, 2004.

[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.