Fitting Linux Device Drivers into an Analyzable Scheduling Framework

[Extended Abstract]

Theodore P. Baker, An-I Andy Wang, Mark J. Stanovich *
Florida State University Tallahassee, Florida 32306-4530
baker@cs.fsu.edu, awang@cs.fsu.edu, stanovic@cs.fsu.edu

ABSTRACT

API extensions and performance improvements to the Linux operating system now enable it to serve as a platform for a range of embedded real-time applications, using fixed-priority preemptive scheduling. Powerful techniques exist for analytical verification of application timing constraints under this scheduling model. However, when the application is layered over an operating system the operating system must be included in the analysis. In particular, the computational workloads due to device drivers and other internal components of the operating system, and the ways they are scheduled, need to match abstract workload models and scheduling polices that are amenable to analysis. This paper assesses the degree to which the effects of device drivers in Linux can now be modeled adequately to admit fixed-priority preemptive schedulability analysis, and what remains to be done to reach that goal.

Categories and Subject Descriptors

D.4.7 [**Software**]: Operating Systems—*organization and design*; C.3.d [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time and embedded systems*

General Terms

design, verification

Keywords

real-time, Linux, fixed-priority scheduling, preemptive, schedulability, device driver

1. INTRODUCTION

A huge amount of theoretical research has been done on real-time scheduling [26]. This theoretical foundation enables one to design a system that can be guaranteed to meet its timing constraints, provided the implementation adheres closely enough to the abstract models of the theory. More specifically, applying the theory requires that the system workload corresponds to models that have been studied, and that the system schedules the workload according to one of the algorithms whose performance on such workloads has been analyzed. Where a real-time system is implemented on top of an operating system, these requirements apply to all the OS components as well as the user-level code.

In Linux and several other POSIX/Unix-compliant [31] operating systems, progress has been made in providing real-time constructs so that user-level programmers can write applications that adhere to the theory of fixed-priority preemptive scheduling. Examples include preemptive priority-based real-time scheduling of user threads, high-precision software timers, and turning off virtual memory management for certain memory regions. Progress also has been made toward making the OS itself adhere more closely to analyzable models, including major reductions in non-preemptible sections within the kernel. Benchmark numbers on existing real-time Linux distributions, such as Montavista [22] and Timesys [32], suggest they now provide adequate capabilities to design and implement a wide range of hard and firm deadline real-time systems at the application level.

However, until recently, the role of device drivers in schedulability has not received much attention. Every operating system includes device drivers, which are responsible for low-level interactions with I/O devices. For embedded real-time systems, device drivers can be especially critical, in two ways. They can play a direct role in meeting throughput requirements and end-to-end deadlines that involve I/O, by the way in which they schedule I/O operations. Device drivers can also play a role in meeting timing constraints for computations that do not depend on I/O, through *interference*; that is, by blocking or preempting more time-critical computations. So, without well-behaved device drivers, the ability of a system to meet timing constraints may be limited to cases where input and output activities do not have deadlines or throughput constraints, and where there are no "storms" of I/O activity. While these are known facts, and while some techniques have been developed for managing I/O performance and device driver interference, integration of that work with Linux is far from mature, and more work remains to be done.

This paper reviews the remaining work to apply fixed-priority preemptive scheduling theory to Linux applications, including the effects of device drivers. It argues that some engineering problems remain to ensure that the interference effects of device drivers fit analyzable models, and to manage device driver scheduling to meet timing constraints, but that the scheduling theory seems adequate. Much larger problems remain with the analysis of I/O scheduling,

*This material is based upon work supported in part by the National Science Foundation under Grant No. 0509131, and a DURIP grant from the Army Research Office.

including device-specific real-time scheduling policies, and end-to-end schedulability analysis involving multiple resources.

2. FIXED-PRIORITY PREEMPTIVE SCHEDULING THEORY

This section reviews some general scheduling theory concepts and terms, and the basic workload models used in fixed-priority preemptive scheduling theory.

The goal of real-time scheduling is to ensure that, if an action is required to execute within a specified time interval it does so. The theory is expressed in terms of *jobs*, *execution times*, *release times*, and *deadlines*. In those terms, the goal is to ensure that each job receives its required execution time within its *scheduling window*, which is the interval between its release time and its deadline.

A job whose own execution time fits within its scheduling window will complete execution within the window unless it is prevented by *interference* from the execution of other jobs. Verifying that a job will be scheduled within its window requires a way to bound the interference, *i.e*, to bound the set of potentially competing jobs and the amount of time that the scheduler will allow them to execute within the window.

A *task* is an abstraction for a stream of jobs, which are ordinarily required to be executed serially with jobs of the same task. Restrictions on the execution times and release times of jobs within each task serve to bound the interference the task can contribute within the scheduling window of another task.

The most analyzed task model is the *periodic task*, in which a task τ_i is characterized by three parameters: the *worst-case execution time*, e_i, of its jobs; the *period*, p_i, between release times; the *relative deadline*, d_i, which is the length of each job's scheduling window. A relaxation of this model is the *sporadic task*, in which the period is interpreted as just a lower bound on the interval between release times. Much is known about the analysis of sets of periodic and sporadic tasks under various scheduling policies.

Fixed-priority preemptive scheduling is very well understood. This theory, including what is sometimes referred to as Generalized Rate Monotonic Analysis (*e.g.*, [15, 1]) and Response Time Analysis (*e.g.*, [3]) makes it possible to verify that a set of hard-deadline tasks will always meet their deadlines, that soft-deadline tasks will satisfy their average response time constraint, and that the execution time of a task may vary within a certain range without causing a missed a deadline.

The foundation of this analysis is a simple *interference bound*, observed by Liu and Layland [19], who showed that a collection of sporadic or periodic tasks causes the maximum amount of interference for a job of lower priority when the job is released together with jobs of all the higher priority tasks and each task releases a job periodically thereafter. It follows that that the interference due to a task τ_i in any interval of length Δ is bounded above by $e_i \lceil \Delta / p_i \rceil$.

Though initially limited to sets of independent preemptible periodic or sporadic tasks with fixed priorities, FP schedulability analysis has been extended to allow for *blocking effects*, due to locks protecting shared resources and brief intervals of increased priority or non-preemptibility due to other causes. In this broader context, there are two ways one task can interfere with another, namely *preemption interference*, based on having higher priority, and *blocking interference*, based on holding a non-preemptible resource that the other task must acquire before it can continue execution.

Fixed-priority preemptive scheduling analysis also has been extended to arbitrary (aperiodic) tasks by assuming that arriving jobs are queued and executed according to an *aperiodic server scheduling policy*. Several aperiodic server scheduling policies have been devised and studied, including the *polling server* [27], the Priority Exchange and Deferrable Server [29, 17], and the Sporadic Server [28]. Without considering the details of specific aperiodic server scheduling algorithms, one can see how they permit schedulability analysis by recognizing that they all enforce the following two principles:

1. Bandwidth limitation: There is an upper bound on the amount of execution time a task may consume (at a given priority) in a given length of time, analogous to the property of a periodic task that it never demands more than e_i time in each interval of length p_i. This permits computation of an upper bound on the amount of preemption interference the aperiodic task can cause for other tasks in an interval of any given length. For the Polling Server and the Sporadic Server with budget e_i the periodic task interference bound applies.

2. Priority bandwidth guarantee: There is a lower bound on the amount of execution time that a thread can rely on being allowed to contend for at a given priority in a given length of time, also analogous to a periodic task. This can generally be translated into a guaranteed average response time to real-time events, and sometimes used to validate hard timing constraints.

3. FIXED-PRIORITY PREEMPTIVE SCHEDULABILITY IN LINUX

This section reviews Linux facilities that support the design and implementation of real-time applications to fit the theory of fixed-priority scheduling, and discusses how well the implementation matches the theory.

Based on the extensive body of knowledge about fixed-priority preemptive scheduling, POSIX/Unix [31] operating systems standards adopted support for scheduling threads at fixed priority (*SCHED_FIFO* and *SCHED_RR*) and via a variation on the Sporadic Server policy (*SCHED_SPORADIC*). Several off-the-shelf operating systems provide support for these policies. Linux currently provides support for the *SCHED_FIFO* and *SCHED_RR* policies. So far, support for the *SCHED_SPORADIC* policy has only been reported in experiments [18], but it will probably eventually appear in Linux distributions.

Application of fixed-priority preemptive scheduling theory in the context of an OS that has no job or task abstractions requires translation between models. The POSIX/Unix API is expressed in terms of threads. A thread is a subprogram that may continue execution indefinitely, alternating between states of contention for execution and self-suspension. To apply the job and task model to a system composed of threads, one needs to treat each point at which a thread suspends itself (*e.g.*, to wait for a timed event or completion of an input or output operation) as the end of a job, and each point at which a thread wakes up from a suspension as the beginning of a new job.

Since the thread model does not constrain the intervals between job releases or the worst-case execution times between suspensions,

systems programmed with threads present problems for schedulability analysis unless some constraints are imposed on thread control flows and/or scheduling policies. Adequate constraints are enforced by the operating system in the case of the *SCHED_SPORADIC* policy, but guaranteeing that the threads scheduled by the *SCHED_RR* and *SCHED_FIFO* policies adhere to an analyzable release time and worst-case execution time model depends on programmer discipline.

Threads that perform input and output operations require additional consideration. If a thread makes blocking I/O requests, the intervals between job release times will depend on both the raw response time of the I/O device and how the system schedules it. For reasons explained in Section 8, the analysis of I/O scheduling, especially in combination with CPU scheduling, is much more difficult than the analysis of CPU scheduling, and is generally beyond the scope of fixed-priority preemptive scheduling theory. A way to work around this limitation is to move I/O out of time-critical threads, so that the CPU and I/O scheduling problems can be modeled and analyzed independently. In Linux, implicit I/O operations due to page fault activity can be avoided in time-critical threads by using *mlock()* and *mlockall()* to lock virtual memory pages accessed by those threads into physical memory. Explicit I/O operations can be moved out by buffering I/O data and either using asynchronous I/O requests, like *aio_read()*, or delegating the I/O to a separately scheduled server thread. Points at which a time-critical thread requires an I/O operation to be completed are deadlines for the I/O scheduler, to be analyzed separately. For example, consider a periodic thread that requires input and produces output, both to the same disk storage device. The input might be requested in advance, with the next task release time as deadline for the input operation, and the output might be buffered, with a deadline for the output operation several times longer than the task period. The scheduling problem is reduced to scheduling three independent periodic tasks, one using just the CPU, one doing disk reads, and one doing disk writes.

It is essential to bound blocking. Any work that is scheduled outside of the fixed-priority preemptive model is a potential source of blocking interference. For analysis to be successful, intervals of time over which a thread may prevent higher priority threads from preempting must have bounded duration. In particular, it is essential to avoid situations where a high-priority thread can wait for a mutex held by a low-priority thread, while a middle-priority thread executes. Linux provides a way to accomplish this, using mutexes with priority inheritance (*PTHREAD_PRIO_INHERIT*).

So far, it appears that the theory of fixed-priority preemptive scheduling can be applied to real-time systems that make use of the POSIX/-Unix thread scheduling policies under an operating system like Linux, provided the user designs the application to fit the models on which the theory is based. The set of real-time (highest) priority threads must be known. Each of them must either use *SCHED_SPORADIC* to limit its maximum high-priority computational demand or use *SCHED_FIFO* or *SCHED_RR* and be verified to fit a well-behaved workload model such as the periodic or sporadic task. In addition, attention must be paid to other details, such as bounding the length of critical sections, and determining bounds on worst-case execution times. All this may be difficult, but it is possible in principle since all of the code is under the user's control.

However, one must also take into account the code of the operating system, which is not directly visible or controllable by a user but may interfere with the schedulability. The OS must fit the models and constraints of the fixed-priority preemptive scheduling theory. Moreover, it is not enough that the OS admit analysis if the analysis does not eventually lead to an application that meets its timing requirements. The OS must admit analysis that is not overly pessimistic, and it must permit an application designer to actively manage priorities and workloads, within the OS as well as at the application level, to meet the timing requirements.

Of course Linux was not originally designed with these goals in mind, and it has since grown so large and complicated that the notion of attempting major architectural changes is daunting. For that reason, some individuals have given up on the idea of using a general-purpose OS like Linux directly as a platform for hard real-time applications, and developed a variety of layered schemes that provide greater control over timing (for example, RTLinux [4, 33], Linux/RK [23], RTAI [6], Hijack [24]). However, at the same time, others have worked to improve the real-time support of the Linux kernel itself (for example, [11, 12].

Probably the biggest improvement has been in bounding blocking effects due to critical sections within the OS. Ingo Molnar [21] introduced a set of high-preemptibility kernel patches, which greatly reduced the average blocking time due to kernel activities. Deriving an exact analytical upper bound for *worst case* blocking still does not seem practical, but an empirical bound can be obtained by measuring the release-time jitter of a periodic thread with the top real-time priority, over a long time and a variety of system loads. Such experiments for recent Linux releases with real-time patches show that blocking interference appears to be bounded [30]. However, in the absence of enforcement, through static or run-time checks, it is possible that a badly written system component could disable preemption for a longer time than observed in the experiments. Worse, unbounded blocking could occur through locking mechanisms, such as Linux kernel semaphores, that neither disable nor implement priority inheritance. Nevertheless, if the probability of blocking exceeding a given empirical bound (and so causing violation of an application timing constraint) can be shown to be low enough, that may be sufficient for many real-time applications.

Given that blocking interference due the OS is bounded, more or less, the remaining challenge is to bound preemption interference. After elimination of the easy cases, by scheduling the system daemons below the real-time priority level, it seems the remaining potential sources of interference by operating system components with the scheduling of application threads are in the system's device drivers.

4. DEVICE DRIVER INTERFERENCE

Device drivers include code that is scheduled in response to hardware interrupts. For example, consider a user task that makes a blocking call to the operating system to request input from a disk drive via a DMA interface. Typically, the driver would execute in three phases, more or less as follows:

1. The client calls the system, and the system calls the device driver. The device driver initiates an input operation on the device, and blocks the client thread until the input operation completes. The device driver code is scheduled as part of the client thread.

2. The device signals completion of the input operation, via an interrupt. An interrupt handler installed by the device driver performs various operations required by the device and input method, such as acknowledging the interrupt and perhaps

copying data from a kernel buffer to a buffer in the client thread's address space, then unblocks the client thread. The scheduling of the device driver code here is *interrupt-driven*.

3. Eventually, execution resumes in the client thread at the point in the device driver code where the client thread blocked. Control flows from the device driver to the kernel and from the kernel back to the user code. While the interrupt from the device plays a role in determining the release time of this phase, the device driver code is scheduled as part of the client thread.

Since the scheduling of interrupt-driven device driver code is outside the direct control of the application, the ability to analyze its effect on the ability of an application to meet timing constraints depends on the design decisions made in the device drivers and operating system kernel.

In popular processor architectures, the hardware schedules interrupt handlers at a priority higher than that of any thread scheduled by the OS[1]. Safe programming practice may also require that an interrupt handler executes non-preemptibly, with interrupts disabled.

In addition, many operating systems schedule interrupt-triggered device-driver code via a two-level mechanism. The Level 1 work, which is executed in interrupt context, is very short; in essence, it just records the fact that the interrupt has occurred, and enqueues the event on a list for later processing. The rest of the work is done at Level 2, in software event handlers.

In Linux, the Level 2 handlers are called *softirq* handlers, though they also go by other names, such "bottom halves" and "tasklets", and "timers". The softirq handlers are executed non-preemptively with respect to the thread scheduler and other softirqs on the same CPU, but with hardware interrupts enabled, in the order they appear in a list. The details of when these handlers are scheduled have changed as the Linux kernel has evolved. As of kernel 2.6.20 the responsibility is divided between two schedulers and priorities. Softirq handlers are executed by *do_softirq()*, which is called typically on the return path from a hardware interrupt handler. If there are still softirqs pending after a certain number of passes through the softirq list (meaning interrupts are coming in fast enough to keep preempting the softirq scheduler), *do_softirq()* returns. Responsibility for continuing execution of softirq handlers is left to be performed at background priority in a scheduled thread (called *ksoftirqd*), or the next time *do_softirq()* is called in response to an interrupt.

Both hardware interrupts and softirq's are intended to provide fast driver response to a particular external event, but can cause problems for schedulability analysis (see Section 6). They can also reduce overall system schedulability. Giving all interrupt-driven work higher priority than all work done by threads introduces a form of priority inversion, where an action that the theory says should logically have higher priority, in order meet its deadline, may be preempted by an action that logically should have lower priority. Executing handlers without preemption introduces another form of priority inversion, where a job that should have higher priority is not able to preempt a job that should have lower priority. Scheduling handlers non-preemptively also introduces a more subtle potential problem, giving up a property of preemptive scheduling that Ha and Liu [10, 9] call *predictability*. This kind of predictability is a necessary basis for schedulability analysis based on just worst-case execution times.

[1]There are systems where interrupts can be assigned hardware priority levels and the CPU interrupt level can be varied, so that hardware interrupt levels can be interleaved with software priority levels. For example, this is possible with the Motorola 68xxx family of processors. It is not clear why this good idea has not been more widely adopted. Perhaps it is one of the many cases of patents on fairly obvious little ideas that impede real technological progress.

Effective scheduling and analysis requires that the use of mechanisms that are exceptions to the overall system scheduling model, such as hardware interrupts and Linux softirqs, be bounded in duration and frequency so that the overall interference they cause can be modeled. The OS can provide mechanisms for drivers to move work into preemptively scheduled threads (see Section 6), but without creative new architectural provisions for responsibility for bounding interrupt handler execution, it must rely on the designer of each device driver to make use of them.

Some interrupt-driven device driver execution presents special problems, due to I/O operations that are *device-driven*. For example, compare the input operations of an Ethernet interface device with disk input operations. An interrupt due to input of a message by a network device can occur spontaneously, due to the nature of an open system, where requests are generated from external sources. In contrast, an interrupt due to completion of a disk operation normally corresponds to a prior request from the kernel or a user thread, reflecting the characteristics of a closed system (overlooking the case in which a network request results in disk activity); so, the frequency of disk requests may be managed by the application, even if the precise timing of the completion interrupts cannot be predicted. Other kinds of input sources that may have device-driven interrupt-scheduled workloads include asynchronous serial ports, and streaming audio and video devices.

Since some portion of the device-driven workload is executed at high priority, it must be bounded before other work can be guaranteed any level of service. For some devices, such as a video device with periodic input behavior, this is not difficult. For other devices, such as an Ethernet interface, one seems to be forced to choose between bounds based on raw hardware capacity, which are unrealistically high, and bounds based on expected worst-case behavior of the communication partners, which cannot be determined by local analysis and may be unreliable. However, the kernel can help by providing aperiodic server scheduling mechanisms that can limit the CPU time spent on some of the interrupt-driven work of a device, as described below in Section 6. Well-designed device drivers may go further, by applying interrupt management techniques, as described in Section 7.

5. DEVICE DRIVER DEMANDS

Device drivers may have timing constraints, such as to stay synchronized with a device or to avoid losing data. For example, consider a video frame grabber (digitizer) device attached to a camera, which inputs video data continuously at a rate of 30 interlaced frames per second. The kinds of timing constraints such a driver might have would depend on the capabilities of the device.

If the frame-grabber requires programmed I/O – *i.e.*, it is not capable of acting as a direct-memory-access (DMA) bus master – the driver must use the CPU to read the device's on-board frame buffer. That will require very close synchronization. The device driver must read raster lines of pixels from the device fast enough to prevent loss of data when the device wraps around and overwrites one frame with the next. A driver designed to capture a full

stream of video data from such a device may be viewed as a periodic task with 1/30-second period. It would have a tight release-time jitter requirement, and a deadline of perhaps half the period, to avoid risk of losing data. If the device generates an interrupt at a known point in each frame, execution of the driver can be driven by that interrupt, but it may need to use another interrupt (via a kernel timer abstraction) to schedule itself at a specified offset from the interrupt. If the device does not generate a frame synchronization interrupt, the driver would need to time its own execution, and the period would probably need to be regulated by the driver to stay in phase with the video source, using a phased-locked-loop control model.

If the frame-grabber is capable of DMA operation, the timing constraints on the driver can be relaxed by providing the device with multiple buffers. The driver may be able to program the device so as to choose which events cause an interrupt to be generated, such as when a new frame has been copied to memory, or when the number of free buffers falls below a threshold. The driver may then be modeled as two virtual tasks: a DMA task (implemented in hardware), and a video driver task (implemented in software) to manage the buffers and synchronization with the consumer of the video data. The DMA task would be periodic and would slow down the CPU by stealing memory cycles from other tasks. If the frame completion interrupt is used, the video driver task can be viewed as a periodic task. Its deadline and jitter requirements would be much looser than the case with programmed I/O, since the additional buffers allow the deadline to be longer than the period. If the threshold interrupt is used, it may be viewed as a sporadic task with a response-time constraint equal to the amount if time it takes the device to consume the number of buffers that is set as the threshold.

Device drivers can have a variety of internal timing constraints. Some cannot be expressed in terms of deadlines, because they are point-to-point within a computation that does not permit giving up control of the CPU. For example, in interactions between the CPU and device there may be a *minimum* delay for the device to process information from the CPU before it is able to accept the next command. If the delay is shorter than the precision of the kernel timer mechanism, achieving adequate throughput may require that the driver busy-wait. There are also point-to-point constraints that dictate non-preemptible execution, such as a *maximum* delay between actions in a sequence of interactions, beyond which the device goes into an error state that requires it to be re-initialized and the entire sequence to be restarted.

In general, device driver internal timing constraints must be validated along with other system timing constraints, and limit the measures that might otherwise be taken to reduce the interference that a device driver causes for other real-time tasks. However, some internal timing constraints that are treated as hard in the design of a driver might better be considered as soft in the context of a particular application. The usual device driver writer's perspective is to treat the needs of the device as top priority. In some cases that is wrong. For example, a device driver writer might decide to disable preemption rather than risk having to reset a device and lose time or data. In the context of an application where the overall function of the device is not time-critical, and some data loss is acceptable, this non-preemptible section might cause a more critical timing constraint to be missed. It is a challenge to design device drivers in a way that provides configuration mechanisms for an application designer to participate in resolving such trade-offs.

The scheduling of device driver execution often imposes a link between the quality of I/O service provided by the driver and the amount of interference the device driver causes for other tasks. It may be blocking interference, such as where disabling preemption within a driver prevents a recoverable device malfunction and loss of data. It may also be preemption interference, at the level of interrupt management and thread scheduling. That is, allowing a device to generate more interrupts or giving higher priority to a device driver thread may allow the device driver to respond to requests for attention from a device, and that may result in less idle time for the device, a shorter response time for the next I/O request to be processed, a higher overall throughput rate, and a reduction in lost data. The right balance in such trade-offs will depend on the application context, so mechanisms are needed for applications designers to configure the scheduling of device-driver execution.

6. DEFERRING WORK TO A THREAD

The less processing is done at hardware interrupt priority the shorter the potential duration of CPU priority inversion, and the better actual system scheduling fits the theoretical ideal. The Linux softirq mechanism might appear to help in this regard, by deferring some of the interrupt-triggered work, but it actually hurts. In contrast to the use of either pure interrupts (generally non-preemptible, top priority) or regularly scheduled (preemptible, lower priority) threads, this kind of complicated mixed scheduling mechanism is very difficult to model and analyze. If one ignores the role of the *ksoftirqd* server, softirqs might be modeled as a per-CPU thread that is scheduled at a fixed priority, lower than the hardware interrupt priority and higher than any other thread. A problem is that the workload of this thread, which is generated by many different kernel components for a great variety of purposes, does not conform to any analyzable model. The demotion of softirq service to be performed at background priority by *ksoftirqd* during heavy bursts of activity helps to bound this load for non-real-time purposes, but it is not done in a way that can be precisely modeled like the well-understood real-time aperiodic server scheduling algorithms.

Clearly, better schedulability can be achieved by moving the work of softirq handlers to one or more regularly scheduled threads. If a device is only used by non-real-time components of the system, the response time of the driver to interrupts will probably not be critical. (This assumption is the basis of the RTLinux [4] practice of deferring all of the Linux interrupt handlers to background processing.) In such cases, it is sufficient to schedule the device server in the background, at low enough priority not to preempt any of the threads that have timing constraints. An exception may be where the device is a bottleneck, but not necessarily, since the techniques described below for reducing interrupts and batching device-driven work by the driver may also result in higher throughput. Of course, if the I/O has throughput or deadline requirements, background scheduling is not sufficient.

It might appear that the interrupt-driven work of device drivers could be moved directly to their client threads, to be scheduled at a priority consistent with the continuation processing after the I/O operation. That does not work very well, for several reasons. One is the technical difficulty of unblocking the client task from inside an interrupt handler without unsafe race conditions. Another reason is that there may be several threads of different priorities sharing access to the same device, so in that case there may be I/O priority inversion, as higher-priority threads with pending I/O requests wait for their requests to be served until the interrupt-driven work of a prior request is executed by a lower priority client. Reducing CPU priority inversion due to interrupt-driven work with-

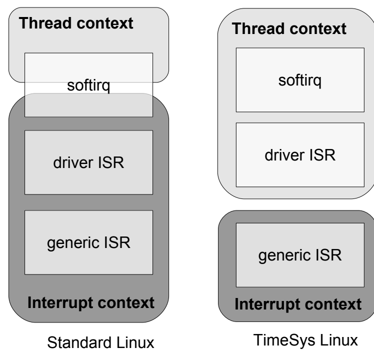

Figure 1: Linux *softirq* handling schemes.

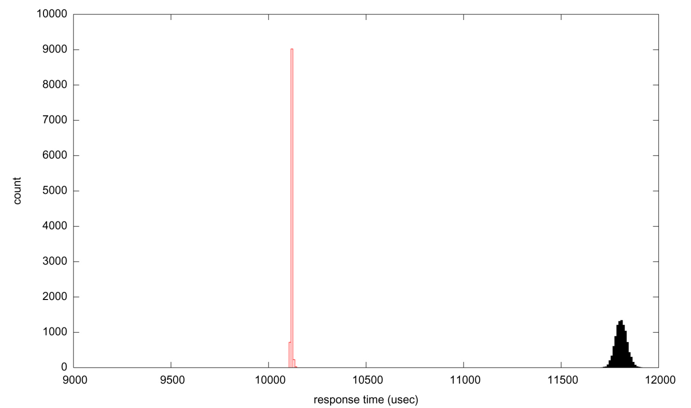

Figure 2: Response time distributions of a task with 100 msec. period and 10 msec. execution time, with and without device driver interference.

out creating such I/O priority inversion can be accomplished by moving work from the interrupt handler to a regularly scheduled server thread of appropriate priority, below interrupt level, but high enough to provide the desired level of I/O service.

Real-time variants of the Linux kernel, including those of Timesys and Montavista, have been modified to execute device-driver interrupt handlers and softirq handlers only from server threads. This is illustrated in Figure 1. For example, in the Timesys kernel, one thread is dedicated to processing interrupts for message-receipt events on the network device, and another to processing message-send events. The priority of each server thread can be set to a level that fits the priority the service it provides. If the priority is lower than that of any real-time thread, preemption interference effects due to softirq's can be ignored in schedulability analysis.

The difference in driver interference effects between running the device-driver server threads at low versus high priority are illustrated in Figure 2. These super-imposed histograms show the response time distribution for a periodic thread, with period of 100 milliseconds and an execution time of 10 milliseconds, running at high real-time priority. I/O load was provided by sending ping packets to the system, at random intervals between 10 and 2000 microseconds, and compiling a Linux kernel at normal user priority. The response times that form a spike between 10 and 10.5 milliseconds are from experiments in which the device driver server threads ran at lower priority than the periodic task. The response times that form a hump between 11.5 and 12 milliseconds are from experiments in which the device driver server threads ran at higher priority.

The idea of using a single interrupt server thread serving multiple interrupts by using a prioritized work queue appears in a 1990 patent by Youngblood [34]. The idea of assigning a thread of appropriate priority to each interrupt appears in a patent by Kleiman in 1996 [14]. The idea of allowing the priority of interrupt server threads to float, at the maximum priority of the set of threads that have devices open that use the corresponding interrupt, appears on the LynxOS 1995 patent, by Bunnell [5]. Regardless of which of these solutions is used, one can model the interrupt-triggered execution of a driver by two tasks, one that has short jobs at interrupt priority, and another that has longer jobs at a lower priority.

Zhang and West [35] proposed a variation of the LynxOS approach. The essential difference is that instead of scheduling the softirq server at the maximum priority of the threads that have an open file description served by the device, they use the maximum priority of the client threads that currently seem to be awaiting service by the device. The processing time for a particular softirq can then be charged against the client thread that it serves. This approach makes sense for device driver execution that can logically be charged to a client thread, but not all I/O has that property. Moreover, it suffers a potential priority inversion problem that is similar to the case if the bottom half were executed directly by the client thread. 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. The network-device server thread would have too low priority 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 finally raise the device server's priority to that of the high priority process. However, since the 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 happen if the low-priority packets were cleared out earlier, as if the device bottom half had been able to preempt the middle-priority task. The LynxOS technique of doing priority inheritance through *open()* operations does not have this problem.

These ideas address the problems of finding the right priority for the interrupt-driven work of the device driver, but they do not address the problem of how to bound the interference due device-driven I/O.

Facchinetti *et al.* [7] recently proposed a way of doing this, but without addressing the priority problem. The system executes all Level 2 interrupt service as one logical thread, at the highest system priority. The thread implements an *ad hoc* aperiodic server scheduling policy, based on budgeting service time at the granularity of individual handler executions. This imposes a bound on the interference the server can cause lower priority threads in any scheduling window. Otherwise, the Level 2 handlers are executed like normal Linux softirq handlers, without preemption by threads

or other Level 2 handlers, in interrupt context, ahead of the application thread scheduler. Since all devices share the same budget and share the same priority, the system does not distinguish different priority levels of I/O, and handles all I/O in FIFO order. This can have undesirable consequences. For example, suppose the network interface is flooded with incoming packets, causing the Level 2 interrupt server thread to exhaust its budget. If a high priority task then requests disk I/O, completion of the disk I/O will be delayed until the Level 2 interrupt sever budget is replenished, and the high priority task may not meet its deadline.

Lewandowski *et al.* [18], proposed a similar approach, but based on using multiple server threads at different priorities, scheduled by an aperiodic server policy at the thread level. They suggest the Sporadic Server policy, since that is already supported for user-level threads by the POSIX/Unix real-time API. This has the virtue of both limiting the interrupt-driven interference that softirqs can cause, regardless of the rate at which the device attempts to push input at the system, while allowing different devices to be served at different priorities, and with different CPU bandwidths. It does not require any modification to existing device drivers.

Lewandowski *et al.* also suggest a way of empirically estimating an upper bound on device driver interference, which can be used directly in schedulability analysis, or used to calibrate the scheduling parameters of a sporadic server.

7. MANAGING INTERRUPTS

Just deferring Level 2 interrupt handling may not be enough. With device-driven input devices, such as a high-speed network interface card, there are situations where the Level 1 hardware interrupt handling alone could cause real-time tasks to miss deadlines. A defense against such an interrupt storm is to disable interrupts. This technique is the basis of the new Linux API for network devices (NAPI) [20], which is implemented by at least one driver. Once an interrupt is received from the device the interrupt is left disabled, at the device level. The Level 2 processing loop, which moves data from the DMA buffers to other buffers and passes them up the protocol stack, runs with the interrupt disabled. It is only re-enabled when the server thread executing the Level 2 loop polls, discovers it has no more work, and so suspends itself.

This mechanism was originally introduced to reduce so-called *receive live-lock*, where a system is so busy handling packet interrupts that it has no time left to process the data, but it has proven to have other benefits. By preventing unnecessary interrupts, it avoids the context-switch overhead for some packets, reducing the net execution time per packet, and so can sustain higher data rates. In addition, when the Level 2 packet processing is done by a thread at low priority level, if packets arrive faster than the server thread can handle them the DMA buffers will fill up and the device will drop packets until the thread has caught up.

The net effect is that interrupts are throttled. A job with higher priority than the Level 2 receive-processing thread can never be preempted by more than one interrupt from the network device. Since the Level 1 interrupt handler is very short, the worst-case interference for high priority tasks is not only bounded, but very small.

The hybrid polling/interrupt technique used in NAPI can be generalized to manage the rate of interrupts from other types of devices. However, barring device malfunctions that cause a stuck interrupt, it should only be needed for devices that are similar to network devices in the sense of spontaneously initiating interrupts. Many other classes of devices will only interrupt to indicate completion of an operation requested earlier by a client, so the rate of interrupts can be managed by a client, by managing the rate of requests.

As time goes on, hardware devices that are capable of generating interrupts at a high rate may provide throttling capabilities directly. That already appears to be the case with the Intel 8254x and 8257x gigabit Ethernet controllers [13], which provide several choices of operating modes in which hardware interrupts may be throttled back to fit a sporadic task model.

Although interrupt throttling ameliorates the problem of interrupt storms, and budgeting time for processing of Level 2 interrupt handling bounds direct interference from the device driver for top priority threads, these methods only indirectly address (via dropped messages) the broader problem of managing the amount of work being accepted into the system. That is, even at acceptable levels of hardware interrupt and softirq activity, some form of early admission control may be needed to throttle the workload of application tasks and to prevent possible resource exhaustion (*e.g.*, buffer space) that might lead to subsequent scheduling interference. Of course, such admission control requires CPU time also, and must be taken into account in the analysis of interference.

8. I/O SCHEDULING EFFECTS

The discussion so far leaves out I/O scheduling, that is, determination of the order and times at which I/O requests are served by each device. Some devices in real-time embedded systems – such as primitive sensors and actuators – do not require I/O scheduling and can perform operations immediately in response to a command from a thread, with no scheduling and very predictable response time. However, there are other I/O devices – such as mass storage devices, network devices, and radars – need scheduling. These are typically devices that need to be shared between threads, have highly variable response times, and may logically perform operations in more than one order. The quality of such I/O scheduling can affect the ability of an application to meet both response time and throughput requirements.

Device drivers may be involved in doing I/O scheduling. They are also affected by I/O scheduling, since the timing and order of I/O operations partially determines the workload of the device driver. Consider a blocking read operating to a disk. The time at which the disk actually performs the operation depends on what other operations are queued for the disk, the order in which they are served, and how long it takes to process each of them.

Schedulablility analysis for I/O is difficult because responsibility for I/O scheduling is distributed among different implementation layers. Device driver software may make some I/O scheduling decisions, but the service order and response times seen at the level of an application task may also be influenced by the device itself and by higher-level system software. For example, while a disk device driver may determine the order in which it passes on the I/O requests it receives, the order in which it receives those requests may be determined by higher-level operating system components, and the order in which the requests it sends out are actually served may be affected by the disk drive itself, by an intermediary controller, a multi-device driver and possibly a logical volume manager in the case of RAID systems, and by filesystem layout. The actual completion time of an I/O request is further complicated by the additional implicit requests for file system metadata associated with the requested I/O. Critical but infrequent error recovery mechanisms

at various levels can also be triggered to perform journal recovery, parity reconstruction, and bad sector forwarding. Full response-time analysis for disk I/O requests will require consideration of the net effect of all these levels. Similarly complex multi-layer interactions are potentially involved in determining service order for other important classes of devices, such as data communication interfaces and radars.

While it may be possible to concentrate the I/O scheduling implementation at one level, there may be a penalty in reduced control over scheduling, or increased overhead and reduced throughput. These are potentially complex trade-offs that need to be resolved for each type of device, and for each application.

Another aspect of I/O scheduling that makes it difficult to analyze is non-preemptivity. Operations on most I/O devices cannot be interrupted, once started. For example, a network interface device would not typically provide the option of interrupting transmission of one message in order start another, nor would that make sense, given the high overhead that such preemption would incur. Similarly, given the long time it takes to get a disk head into position to access a given sector, it would not makes sense to preempt a disk drive in the midst of an I/O operation. As mentioned above, one consequence of non-preemptivity is that the scheduling effects of execution time variation in cannot be bounded by just considering the shortest and longest cases [10, 9].

The difficulty of even single-device analysis is exacerbated by the fact that I/O times can be context-sensitive. For example, in the case of a disk drive the time to access a given block depends on the position of the disk head relative to the block location and the content of the driver's local cache. That, in turn, depends on what operation was scheduled before.

Another important issue that impacts schedulability analysis involving I/O response times is timing constraints that span multiple jobs, involving several different processors and precedence constraints. For instance, a network video server might read video frames from a frame grabber or a local disk, perform some computation on the video data (say trans-coding or frame skipping), and transport the requested data across the network. In this example, there is a precedence ordering among jobs on the different devices, and each job depends upon the successful and timely completion of a previous task in the sequence. In addition, to achieve a reasonable perceptual quality, the entire sequence of tasks needs to be repeated regularly.

I/O scheduling also can involve multiple conflicting quality–of-service criteria. For example, priority-based scheduling of disk I/O can reduce response time for a few high-priority requests, but at the cost of increased total processing time (for head movement and rotational latency), which means reduced throughput. Algorithms that provide good average throughput provide very hard-to-predict response times. So, if a system has tasks with both response time and throughput requirements, perhaps in the same task, it is not clear what to do.

For the reasons given above and others, when I/O scheduling is considered together with scheduling of the CPU and possibly other devices, the analysis problem becomes extremely difficult. The theory, so far, has very little useful to say about these problems. Some research has been done on limited aspects of the end-to-end I/O scheduling problem (*e.g.*, [2, 16, 25, 8, 36]), from the perspective of either resource allocation to support QoS or co-ordinated scheduling of specific set of resources. However, a comprehensive theory of multi-resource allocation and schedulability analysis does not yet seem to exist.

9. CONCLUSION

The state of practice in Linux seems close to providing adequate support for constructing a variety of real-time systems to meet timing constraints by design, and verifying them, based on preemptive scheduling theory. However, there remains some work to be done at the level of device drivers.

One of the advantages of a mature, widely used, general-purpose operating system like Linux is that it already has a large collection of device drivers. There are good reasons for trying to reuse some of these drivers in a real-time application. Techniques exist that permit modeling the role of some existing device drivers in system schedulability, including ways of bounding the interference real-time application tasks may experience due to the Level 2 interrupt processing, with only minor changes to the way softirq handing is scheduled by the kernel. With further change, in the design of device drivers, the interference due to Level 1 interrupt processing may also be bounded.

These solutions do require tailoring of how Level 2 interrupt handling is scheduled and how Level 1 interrupts are throttled, to fit the needs of an application. That currently can only be done by modification of the kernel and/or device driver code. It is possible in an open-source system, but is still an obstacle to widespread use. For the adoption of such techniques with proprietary operating systems, it is a more significant roadblock.

This situation could be improved by expanding the device driver and user programming interfaces, to provide more visibility and control over device driver scheduling and interrupt handling at the application level.

It may also be advantageous to provide enforcement mechanisms to improve the real-time quality of device drivers, which are developed independently by a large number of individuals, who are not all fully aware of what effects their device driver might have on the schedulability of other system components, or the relative importance of driver-internal timing constraints as compared to other requirements in a particular real-time application.

Problems that are more serious exist with regard to achieving and verifying end-to-end timing requirements that span I/O operations, which need to be solved better in theory before one can talk seriously about how to support the theory better in an operating system. A reasonable way to make progress is to look at restricted cases, such as computations involving just the CPU and one I/O resource, either modeling existing Linux I/O scheduling policies for the device or modifying the device scheduling policy to make real-time performance more analyzable.

REFERENCES

[1] Advanced Informatics. SCAN-schedulability analysis tool. `http://www.spacetools.com/tools4/space/272.htm`.

[2] D. P. Anderson. Meta-scheduling for distributed continuous media. Technical Report CSD-90-599, ECE Department, University of California at Berkeley, Dec. 1990.

[3] 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.

[4] M. Barabanov. A Linux-based real-time operating system. Master's thesis, New Mexico Instituted of Techology, Albuquerque, NM, June 1997.

[5] M. Bunnell. Operating system architecture using multiple priority light weight kernel task based interrupt handling, u. s. patent 5,469,572. `http://www.upsto.gov`, 1995.

[6] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI position paper, nov 2000. In *RTSS 2000 Real-Time Operating Systems Workshop*. IEEE Computer Society, 2000.

[7] 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.

[8] K. Gopalan and T. Chiueh. Multi-resource allocation and scheduling for periodic soft real-time applications. In *Proc. Multimedia Computing and Networking*, San Jose, CA, USA, Jan. 2002.

[9] R. Ha. *Validating timing constraints in multiprocessor and distributed systems*. PhD thesis, University of Illinois, Dept. of Computer Science, Urbana-Champaign, IL, 1995.

[10] R. Ha and J. W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. 14th IEEE International Conf. Distributed Computing Systems*, pages 162–171, Poznan, Poland, June 1994. IEEE Computer Society.

[11] A. C. Heursch, D. Grambow, A. Hosrtkotte, and H. Rzehak. Steps towards a fully preemptable Linux kernel. In *Proc. 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming*, Lagow, Poland, May 2003.

[12] A. C. Heursch, D. Grambow, D. Roedel, and H. Rzehak. Time-critical tasks in Linux 2.6: Concepts to increase the preemptability of the Linux kernel. In *Linux Automation Konferenz*, Germany, Mar. 2004. University of Hanover.

[13] Intel Corporation. Interrupt moderation using intel gigabit ethernet controllers (AP-450). Application Note, Intel Corporation, Apr. 2007.

[14] S. Kleiman. Apparatus and method for interrupt handling in a multi-threaded operating system kernel. U. S. Patent 5,515,538, 1996.

[15] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practioner's Handbook for Real Time Analysis: Guide to Rate Monotonic Analysis for Real Time Systems*. Kluwer, Boston-Dordrecht-London, 1993.

[16] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource QoS problem. In *Proc. IEEE Real-Time Systems Symposium*, Phoenix, AZ, USA, Dec. 1999.

[17] 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.

[18] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 57–68, Bellevue, WA, Apr. 2007.

[19] 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.

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

[21] I. Molnar. Preemptive kernel patches. `http://people.redhat.com/mingo/`.

[22] Montavista, Inc. Montavista Linux. `http://www.mvista.com`.

[23] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in linux. In *Real-Time Systems Symposium Work-in-Progress*, Dec. 1998.

[24] G. Palmer and R. West. Hijack: Taking control of cots systems for real-time user-level services. In *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 133–146, Bellevue, Washington, Apr. 2007. IEEE Computer Society Press.

[25] S. Saewong and R. Rajkumar. Cooperative scheduling of multiple resources. In *Proc. IEEE Real-Time Systems Symposium*, Phoenix, AZ, USA, Dec. 1999.

[26] L. Sha, T. Abdelzaher, K. E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2–3):101–155, Nov. 2004.

[27] 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.

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

[29] 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.

[30] G. H. Thaker. Real-time OS periodic tests. `http://www.atl.external.lmco.com/projects/QoS/RTOS_html/periodic.html`.

[31] The Open Group. The single Unix specification, version 3. `http://www.unix.org/version3/`.

[32] TimeSys, Inc. Embedded Linux development products. `http://www.timesys.com`.

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

[34] L. Youngblood. Interrupt driven prioritized queue. U. S. Patent 4,980,820, 1990.

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

[36] Y. Zhou and H. Sethu. On achieving fairness in the joint allocation of processing and bandwidth resources: Principles and algorithms. Technical Report DU-CS-03-02, Drexel University, 2003.