Supporting Time-Sensitive Applications on General-Purpose Operating Systems

Ashvin Goel, Luca Abeni, Jim Snow, Charles Krasic, Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute, Portland
{ashvin,luca,jsnow,krasic,walpole}@cse.ogi.edu

Abstract

General-purpose operating systems are increasingly being used for serving time-sensitive applications. Supporting these applications requires low-latency response from the kernel and from other system-level services. This paper explores various operating systems techniques needed to support time-sensitive applications and describes the design of a time-sensitive, general-purpose Linux system. We show that a high-precision timing facility together with a well-designed preemptible kernel can be the basis for a low-latency response system and such a system can have low overhead. We evaluate the behavior of realistic time-sensitive user- and kernel-level applications on our system and show that, in practice, it is possible to properly support time-sensitive applications in a general-purpose operating system without compromising the performance of throughput-oriented applications.

1 Introduction

Multimedia applications, and soft real-time applications in general, are driven by real-world demands and are characterized by timing constraints that must be satisfied for correct operation; for this reason, we call these applications *time-sensitive*. Time-sensitive applications may require, for example, periodic execution with low jitter, or response in a short period of time to external events such as the arrival of network packets.

To support time-sensitive applications, a general-purpose operating system must respect the application's timing constraints. For these constraints to be satisfied, resources must be allocated to the application at the appropriate times, hence resource allocation must be accurate.

As we will show in Section 2, there are some important requirements for achieving a correct resource allocation; each of these requirements have been addressed in the past with specific mechanisms, but unfortunately general-purpose operating systems, such as Linux, often do not support or integrate these mechanisms.

This paper focuses on three specific techniques that can be integrated to satisfy the constraints of time-sensitive applications. First, we present *firm timers*, an efficient high-resolution timer mechanism.

Firm timers incorporate the benefits of periodic timers, one-shot timers available on modern hardware [8] and soft timers [5] and provide accurate timing with low overhead. Second, we use a preemptible kernel design for a responsive kernel. Finally, we use both priority and reservation-based CPU scheduling mechanisms for supporting various types of time-sensitive applications. We have integrated these techniques in our extended version of the Linux kernel, which we call *Time-Sensitive Linux* (TSL).

Currently, general-purpose systems provide coarse-grained resource allocation with the goal of maximizing system throughput. Such a policy conflicts with the needs of time-sensitive applications which require more precise allocation. Thus, recently several approaches have been proposed to improve the timing response of a general-purpose system such as Linux [17, 24]. These approaches include improved kernel preemptibility and a more generic scheduling framework. Since their focus is hard real-time, they do not evaluate the performance of non-real time applications.

In contrast TSL focuses on integrating an efficient support for time-sensitive applications in a general-purpose

This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs and by Intel.

OS without degrading the performance of traditional applications. One of the main contribution of this paper is to show through experimental evaluation that using the above techniques it is possible to provide good performance to time-sensitive applications as well as to throughput-oriented applications.

The rest of the paper is organized as follows. Section 2 investigates the factors that contribute to poor temporal response in general-purpose systems. Section 3 describes the techniques that we have used to implement our time-sensitive Linux system. Section 4 evaluates the behavior of several timing-sensitive applications and presents overheads of our time-sensitive Linux system. Finally, in Section 5 we state our conclusions.

1.1 Related Work

The scheduling problem has been extensively studied by the real-time community [11, 14, 20]. However, most of the scheduling analysis is based on an abstract mathematical model that ignores practical systems issues such as kernel non-preemptibility and interrupt processing overhead. Recently, many different real-time algorithms have been implemented in Linux and in other general purpose kernels. For example, Linux/RK [17] implements Resource Reservations in the Linux kernel, and RED Linux [24] provides a generic scheduling framework for implementing different real-time scheduling algorithms. These kernels tackle the practical systems issues mentioned above with techniques similar to the techniques presented in this paper. For example, RED Linux inserts preemption points in the kernel, and Timesys Linux/RT (based on RK technology) uses kernel preemptibility for reducing kernel latency. Kernel preemptibility is also used by MontaVista Linux [1]. However, while these kernels work well for time-sensitive applications, their performance overhead on throughput-oriented applications is not clear.

A different approach for providing real-time performance is used by other systems, such as RTLinux [6], RTAI [13], and KURT[21], which decrease the unpredictability of the system by running Linux as a background process over a small real-time executive. In this case, real-time tasks are not Linux processes, but run on the lower-level real-time executive, and the Linux kernel runs as a non real-time task. This solution provides good real-time performance, but does not provide it to Linux applications. Linux processes are still non real-time, and cannot support time-sensitive applications. Also, native real-time threads use a completely different, and less evolved, ABI compared to the Linux one, and do not have access to Linux device drivers.

An accurate timing mechanism is crucial for supporting time-sensitive applications. Thus most of the existing real-time kernels or real-time extensions to Linux provide high resolution timers. The high resolution timers concept was proposed by RT-Mach [19] and has subsequently been used by several other systems such as Rialto [9]. In a general-purpose operating system, the overhead of such timers can affect the performance of throughput-oriented applications. This overhead is caused by the increased number of interrupts generated by the timing mechanism and can be mitigated by the soft-timer mechanism [5]. Thus, our firm-timer implementation uses soft timers.

Finally, the Nemesis operating system [10] is designed for multimedia and other time-sensitive applications. However, its structure and API is very different from the standard programming environment provided by general-purpose operating systems such as Linux. Our goal is to minimize changes to the programming environment to encourage the use of time-sensitive applications in a general-purpose environment.

2 Time-Sensitive Requirements

As said, to respect applications' temporal constraints resources must be allocated at the appropriate times, where the appropriate times are determined by events of interest to the application, such as readiness of a video frame for display. Hence, we can view the timeline of the application as a sequence of such *events* and the corresponding *application's activations*. For example, Figure 1 shows an event and the corresponding activation. As the figure shows, there is a latency between the event and the activation. This latency has three components called *timer resolution latency*, *non-preemptible section latency* and *scheduling latency* as shown in the figure, which depicts the execution sequence in a system after a wall-clock time event. A time-sensitive system needs low total latency.

There are three requirements for providing allocations with low latency: 1) an accurate timing mechanism, 2) a responsive kernel and 3) an appropriate CPU scheduling algorithm. These requirements are described below.

Timing Mechanism: An accurate timing mechanism is crucial for reducing latency as timer resolution is

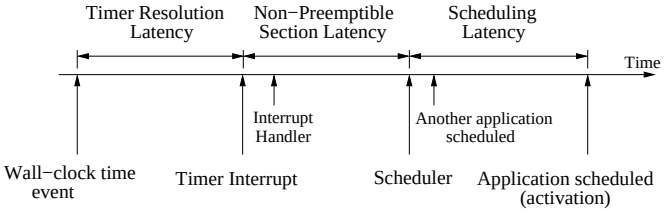


Figure 1: Execution sequence after wall-clock time expiration event.

the most common source of latency in an operating system such as Linux [4]. Such a mechanism can be implemented efficiently by using two techniques, one-shot timers available on most common modern hardware and soft timers [5]. These techniques are complementary and can be combined together. One-shot timers can provide high accuracy, but, unlike periodic timers, they require reprogramming at each activation. On an x86 machine, one-shot timers can be generated using the on-chip CPU Advanced Programmable Interrupt Controller (APIC). This timer has very high resolution and can be reprogrammed in a few CPU cycles.[1]

Soft timers check and run expired timers at strategic points in the kernel thus reducing the number of hardware generated timer interrupts and the number of user-kernel context switches. We call the combination of these two mechanisms, *firm timers*. In Section 4.3, we show that the overhead of firm timers is not significant.

Responsive Kernel: An accurate timing mechanism is not sufficient for reducing latency. For example, even if a timer interrupt is generated by the hardware at the correct time, the activation could occur much later because the kernel is unable to interrupt its current activity. This problem occurs because either the interrupt might be disabled or the kernel is in a non-preemptible section. In traditional kernels, a thread entering the kernel becomes non-preemptible and must either yield the CPU or exit the kernel before an activation can occur. The solution is to reduce the size of such non-preemptible sections as described in Section 3.2.

CPU Scheduling Algorithm: The scheduling problem for providing precise allocations has been extensively studied in the literature but most of the work relies on some strict assumptions such as full preemptibility of tasks. A responsive kernel with an accurate timing mechanism enables implementation of such CPU scheduling strategies because it makes the assumptions more realistic and improves the accuracy of scheduling analysis. In this paper, we use two different real-time scheduling algorithms: a proportion-period scheduler and a priority-based scheduler.

The proportion-period scheduler provides temporal protection to tasks. With proportion-period, we model application behavior by identifying a characteristic delay that the application can tolerate and allocating a fixed proportion of the CPU within a period equal to the delay to each task in the application. Alternatively, we assign priorities to all tasks in some application-specific order for use with the priority scheduler.

While each of these requirements have been addressed in the past, they have generally been applied to specific problems in limited environments. When applied in isolation in the context of a general-purpose system, they fail to provide good time-sensitive performance. For example, a high resolution timer mechanism is not useful to user-level applications without a responsive kernel. This probably explains why soft timers [5] did not export their functionality to the user level through the standard POSIX API. Conversely, a responsive kernel without accurate timing has only a few applications. For example, the low latency Linux kernel [16] provides low latency only when an external interrupt source such as an audio card is used.

Similarly, a scheduler that provides good theoretical guarantees is not effective when the kernel is not responsive or its timers are not accurate. Conversely, a responsive kernel with an accurate timing mechanism is unable to handle a large class of time-sensitive applications without an effective scheduler. Unfortunately, these solutions have generally not been integrated: on one hand, real-time research has developed good schedulers and analyzed them from a mathematical point of view, and on the other hand, there are real systems that provide a responsive kernel but provide simplistic schedulers that are only designed to be fast and efficient [1]. Real-time operating systems integrate these solutions for time-sensitive tasks but tend to ignore the performance overhead of their

[1] In contrast, reprogramming the standard programmable interval timer (PIT) on an x86 is very expensive because it requires several slow `out` instructions on the ISA bus.

solutions on throughput-oriented applications [17]. Our goal is to support both types of applications well.

It is worth noting that latency due to system services, such as the X11 server for graphical display [3] on a Linux system, has the same components as kernel latency described above. In fact, the simple scheduling model presented above assumes that tasks are independent. In a real application, tasks can be interdependent which can cause priority inversion problems. For example, in Section 4.2.1 we show that a multimedia player that uses the X11 server for display can perform sub-optimally due to priority inversion, even if the kernel allocates resources correctly. The X11 server operates on client requests in an event-driven manner and the handling of each event is non-preemptible and generally in FIFO order. As a result, time-sensitive clients expecting service from the server observe latencies that depend on the time to service previous client requests: the performance of time-sensitive applications depends on not just kernel support for such applications but also the design of other system services. Thus in Section 3.3 we enhance the priority scheduler with techniques that solve priority inversion.

In the next section, we present the components of TSL, providing accurate timers, a responsive kernel, and time-sensitive scheduling algorithms to support the requirements highlighted above.

3 Implementing Time-Sensitive Systems

We propose three techniques, firm timers, kernel preemptibility and proportion-period CPU scheduling that aim to reduce the total latency. We have integrated these techniques in the 2.4.16 Linux kernel to implement TSL.

3.1 Firm Timers

Firm timers provide an accurate timing mechanism with low overhead by combining three approaches for implementing timers: one-shot timers, periodic timers and soft timers. One-shot timers provide high accuracy while periodic timers and soft timers provide low overhead timing.

Periodic timers are normally implemented with periodic tick interrupts. For example, on x86 machines, this interrupt is generated by the Programmable Interval Timer (PIT), and on Linux, the period of the interrupt is 10 ms. As a result, the maximum timer resolution latency is 10 ms. This latency can be reduced by reducing the tick period but this solution increases system overhead because more tick interrupts are generated. To reduce the overhead of timers, we have to move from a periodic timer interrupt model to a one-shot timer interrupt model where interrupts are generated only when needed. The following example explains the benefits of one-shot interrupts. Consider two tasks with periods 5 and 7 ms. With periodic interrupts, the tick period must be 1 ms to eliminate timer resolution latency. Hence in 35 ms, there would be 35 interrupts generated. With one-shot interrupts, interrupts will be generated at 5 ms, 7 ms, 10 ms, etc., and the total number of interrupts is 11. One-shot timers have to be reprogrammed for the next timer event at each activation (unlike periodic timers) but avoid the flood of unwanted interrupts that are necessary to maintain good accuracy with a periodic timer and an aperiodic interval.

One-shot timers improve timer accuracy but have their own sources of overhead. These overheads occur due to three reasons: 1) reprogramming at each activation, 2) maintenance of timer data structures, and 3) increased interrupts with fine-grained timing. The data structures for one-shot timers are less efficient than for periodic timers. For instance, periodic timers can be implemented using calendar queues [7] which operate in $O(1)$ time, while one-shot timers require priority heaps which require $O(log(n))$ time, where n is the number of active timers. This difference exists because periodic timers have a natural bucket width (in time) that is the tick period of the timer interrupt. Calendar queues need this bucket width and derive their efficiency by providing no ordering to timers within a bucket. One-shot fine-grained timers have no corresponding bucket width. Firm timers are implemented using one-shot timers. Below, we describe the methods used by firm timers to reduce each source of overhead.

Timer reprogramming: Fortunately, timer reprogramming is inexpensive on modern hardware. For example, our firm-timers implementation uses the APIC one-shot timer present in modern x86 machines. The APIC is set by writing a value into a register which is decremented at each memory bus cycle until it reaches zero and generates an interrupt. Given a 100 mhz memory bus, a one-shot timer has a theoretical accuracy of 10 nanoseconds.[2] This

[2] In practice, the interrupt handler is much slower than 10 ns and is the limiting factor for timer accuracy.

timer resides on-chip an can be reprogrammed in a few cycles without noticeable performance penalty.

Data structures: Firm timers maintain a timer queue for each processor. When the APIC timer expires, the interrupt handler checks the timer queue and executes the callback function associated with each expired timer in the queue. Expired timers are removed while periodic timers are re-enqueued after their expiration field is incremented by the value in their period field. The APIC timer is then reprogrammed to generate an interrupt at the next timer event.

The firm timer expiration times are specified as CPU clock cycle values. We use two performance optimizations in the firm timers implementation. First, the expiration time is specified as a 32 bit quantity although the current time in CPU cycles is stored in a 64 bit register in an x86 processor. The 32 bit expiration value avoids expensive 64 bit time conversions from CPU cycles to memory cycles needed for programming the APIC timer. This choice of a (signed) 32 bit value limits the use of firm timers to a maximum one second timeout on a modern two Ghz processor due to time roll over. Fortunately, one-shot timers are not needed for long timeouts. Instead, and second, firm timers combine periodic timers together with one-shot timers for long timeouts. A firm timer for a long timeout uses a periodic timer to wake up at the last period before the timer expiration and then sets the one-shot APIC timer. Hence, our firm timers approach only has active one-shot timers within one tick period. Since the number of such timers, n, is decreased, the data structure implementation is more efficient.

Increased interrupts: The increased timer accuracy of one-shot timers comes at the cost of increased overhead because every timer interrupt leads to a user-kernel context switch that can stall the CPU pipeline and can result in cache pollution. To solve this problem, firm timers use soft timers [5]. The APIC timer can be set to always overshoot by a fixed amount of time called *timer latency*. Soft timers add checks for expired timers at strategic points in the kernel such as system call, interrupt, and exception return paths. In some cases, an interrupt, system call, or exception may happen after a timer has expired but before the APIC timer generates an interrupt. At this point, the timer expiration is handled and the APIC timer is reprogrammed for the next timer event. This approach avoids the extra user-kernel context switch overhead of the timer interrupt at the cost of some timer accuracy.

Soft timer checks are normally placed at kernel exit points, which end critical sections in the kernel and where the scheduler function can be invoked. A preemptible kernel design, as described in Section 3.2, reduces the granularity of critical sections in the kernel and thus allows more frequent soft timer checks and hence can provide better timing accuracy.

The timer overshoot or latency value allows making a tradeoff between accuracy and overhead. A small value of timer latency provides high timer resolution while a large value decreases the timer overhead. This latency value can be changed dynamically. With a zero value, we obtain hard timers and with a large value, we obtain soft timers. A choice in between leads to a hybrid approach that is used by our firm timers. This choice depends on the timing accuracy needed by the application. Our current implementation uses a single global latency value but it is easy to extend this implementation so that each timer can specify its desired latency.

We want to provide the benefits of the accurate timing mechanism to standard user-level applications. These applications use the standard POSIX interface calls such as `nanosleep()`, `pause()`, `setitimer()`, `select()` and `poll()`. We have modified the implementation of these system calls to use firm timers without changing their interface. As a result, unmodified applications automatically get increased timer accuracy in our system.

3.2 Kernel Responsiveness

A kernel is responsive when its non-preemptible sections, which keep the scheduler from being invoked to schedule a task, are small. For example, if the timer interrupt in Figure 1 is disabled, the task can only enter the ready queue when the interrupt is re-enabled. In addition, the task upon entering the ready queue may still not be scheduled if another task is running in the kernel in a non-preemptible section.

The length of non-preemptible sections in a kernel depends on the strategy that the kernel uses to guarantee the

consistency of its internal structures and on the internal organization of the kernel. Traditional general purpose kernels allow at most one execution flow in the kernel at any given time by disabling preemption when an execution flow enters the kernel, i.e., when an interrupt fires or when a system call is invoked. Thus the non-preemptible section latency is equal to the maximum length of a system call plus the processing time of all the interrupts that fire before returning to user mode. Unfortunately, this value can be as large as 100 ms [4].

One approach that reduces non-preemptible section latency is explicit insertion of preemption points at strategic points inside the kernel so that a thread in the kernel explicitly yields the CPU to some other thread after it has executed for some time. In this way, the size of non-preemptible sections is reduced. The choice of preemption points depends on system call paths and has to be manually placed after careful auditing of system code. This approach is used by some real-time versions of Linux, such as RED Linux [17] and by Andrew Morton's low-latency project [16]. Non-preemptible section latency in such a kernel decreases to the maximum time between two preemption points.

Another approach, used in most real-time systems, removes the constraint of a single execution flow inside the kernel. Thus it is not necessary to disable preemption when an execution flow enters the kernel. To support this level of kernel preemptibility, kernel data must be explicitly protected using mutexes or spinlocks. The Linux preemptible kernel project [12] uses this approach and disables kernel preemption only when a spinlock is held. In a preemptible kernel, the non-preemptible section latency is determined by the maximum amount of time for which a spinlock is held inside the kernel.

Our previous evaluation [4] shows that these approaches work fairly well and should be incorporated in the design of any time-sensitive kernel. Our experiments with real applications in Section 4.2 show that a responsive kernel complements an accurate timing mechanism to help improve time-sensitive application performance.

3.3 CPU Scheduling

The CPU scheduling algorithm should ensure that time-sensitive tasks obtain their correct allocation with low latency. We use the proportion-period and priority models as described Section 2 to schedule time-sensitive applications. The proportion-period model provides temporal protection to applications and allows balancing the needs of time-sensitive applications with non-real time applications but requires specification of proportion and period scheduling parameters of each task. The priority model has a simpler programming interface but assumes that the timing needs of tasks are well-behaved.

3.3.1 Proportion-Period CPU Scheduling

For a single independent task, the simplest scheduling solution is to assign the highest priority to the task. However, with this solution, a misbehaving task that does not yield the CPU can starve all other tasks in the system. A time-sensitive general purpose system should provide *temporal protection* to tasks so that misbehaved tasks that consume "too much" execution time do not affect the schedule of other tasks. The temporal protection property is similar to memory protection in standard operating systems that provides memory isolation to each application.

Our proportion-period allocation model automatically provides temporal protection because each task is allocated a fixed proportion every period. The period of a task is related to some application-level delay requirement of the application, such as the period of a periodic task, or it can be derived from the jitter requirements of a time-sensitive task. The proportion is the amount of CPU allocation required every period for correct task execution. The proportion-period model can be effectively implemented using well known results from real-time scheduling research[15]. In this implementation, classical real-time scheduling techniques (EDF or RM priority assignment) are used and a task is allowed to execute as a real-time task for a time Q equal to the product of its proportion P and its period T and then blocking the task (or scheduling it as a non real-time task) until its next period. In this way, a task is *reshaped* so that it behaves like a periodic real-time task with parameters (Q, T) and can be properly scheduled by a classical real-time scheduler. A similar technique is used in networks by traffic shapers such as leaky buckets or token buckets.

We have implemented a proportion-period CPU scheduler in Linux to provide temporal protection to tasks. This scheduler uses an EDF scheduling mechanism to obtain full processor utilization. In addition, when two tasks have the same deadline, the one with the smallest remaining capacity is scheduled to reduce the average finishing time.

3.3.2 Priority CPU Scheduling

In the priority model, real-time priorities are assigned to time-sensitive tasks based on application needs [11]. One key problem with the priority model is priority inversion which occurs when an application is composed of multiple tasks that are interdependent. The classic priority inversion problem occurs with three tasks. For example, consider a simple example of a video application consisting of a client and an X11 server. Let us assume that the client has been assigned the highest priority because it is time-sensitive. It displays graphics by requesting services from the X11 server. When it needs to display a video frame, it sends the frame to the server and then blocks waiting for the display to complete. If the X11 server has a priority lower than the client's priority, then it can be preempted by another task with a medium priority. Hence the medium priority task is delaying the server and thus delaying the high-priority client task.

We use a variant of the priority ceiling protocol [20] called the highest locking priority (HLP) protocol to cope with priority inversion. The HLP protocol works as follows: when a task acquires a resource, it automatically gets the highest priority of any task that can acquire this resource. In the example above, the display is the shared resource and thus the X11 server gets the highest priority among all time-sensitive clients accessing it. Hence, the X11 server gets the priority of the client task and is not preempted by the medium priority task.

The HLP protocol is very general and works with across multiple servers. Interestingly, this protocol handles the FIFO ordering problem in server queues mentioned in Section 3. Since servers have the highest priority among all their potential clients, they are able to serve each request immediately after it is enqueued and thus the queue size is never more than one and the queuing strategy is not relevant. After servicing the request, the next highest-priority client is scheduled and the latency caused by the server is minimized.

4 Evaluation

This section describes the results of experiments we performed to evaluate the behavior of time-sensitive applications running on this system and the overheads of this system. Our experiments focus on evaluating the behavior of realistic time-sensitive applications running on a loaded general-purpose environment, and were run on a 1.5 Ghz Pentium-4 Intel processor with 512 MB of memory.

4.1 Micro Benchmarks

Before evaluating the impact of the latency reduction techniques used in TSL on a real application, we performed some micro-benchmarks for evaluating the kernel latency as defined in Section 2.

As previously shown, the kernel latency is composed by the timer resolution latency, the non-preemptible latency, and the scheduling latency. We evaluated the first two components in isolation by running a time-sensitive process that requires to sleep for a specified amount of time (using the `nanosleep()` system call) and measures the time that it actually sleeps. In a first set of experiments, we evaluated the timer resolution latency, showing that it is $10ms$ in standard Linux, and that firm timers can reduce it to few microseconds.

After that, we evaluated the non-preemptible latency when a number of different system loads are ran in background. The first interesting result was that on standard Linux the worst case non-preemptible latency (occurring when the kernel is copying large amounts of data between kernel and user space) can arrive to $100ms$, but that in most common cases the non-preemptible latency is less than $10ms$, and it is hidden by the timer resolution latency. However, when firm timers are used the non-preemptible latency becomes more visible, and it is possible to see that it is easy to obtain latencies bigger than $5ms$. Using appropriate kernel preemptibility techniques, the latency can be greatly reduced, and TSL provides a maximum kernel latency of less than $1ms$ on our test machine.

The full details of the experiments and more results are presented in our previous paper [4], that we briefly summarized here for the reader's convenience.

4.2 Latency of Real Applications

After evaluating the kernel latency in isolation through micro-benchmarks, we performed experiments on two real applications, mplayer and the proportion-period scheduler which is a kernel-level application. We choose audio/video synchronization skew as the latency metric for mplayer. The latency metric for the proportion-period scheduler is maximum error in the allocation and period boundary.

4.2.1 Mplayer

Mplayer [2] is an audio/video player that can handle several different media formats. Mplayer synchronizes audio and video streams by using timestamps that are associated with the audio and video frames. The audio card is used as a timing source, i.e., audio samples are put in the audio card buffer, and when a video frame is decoded, its timestamp is compared with the timestamp of the currently played audio sample. If the video timestamp is smaller than the audio timestamp then the program is late (i.e., a video deadline has been missed) and the video is immediately displayed. Otherwise, the system sleeps until the video and audio timestamps are equal and then displays the video.

On a responsive kernel and with sufficient available CPU capacity, audio/video synchronization can be achieved by simply sleeping for the correct amount of time. Thus, mplayer uses the Linux `nanosleep()` call for synchronization. Unfortunately, if the kernel is unresponsive, mplayer will not be able to sleep for the correct amount of time leading to poor audio/video synchronization and high jitter in the inter-frame display times. Synchronization skew and display jitter are correlated and thus this paper presents results only for audio/video synchronization skew.

We compare the audio/video skew of mplayer on standard Linux and on our time-sensitive Linux under three competing loads: 1) non-kernel CPU load, 2) kernel CPU load, and 3) file system load. For non-kernel load, a user-level CPU stress test is run in the background. For kernel CPU load, a large memory buffer is copied to a file, where the kernel uses CPU to move the data from the user to the kernel space. Standard Linux does this activity in a non-preemptible section. This load spends 90% of its execution time in kernel mode. For the file system load, a large directory is copied recursively and the file system is flushed multiple times to create heavy file system activity. In each of these tests, mplayer is run for 90 seconds at real-time priority.

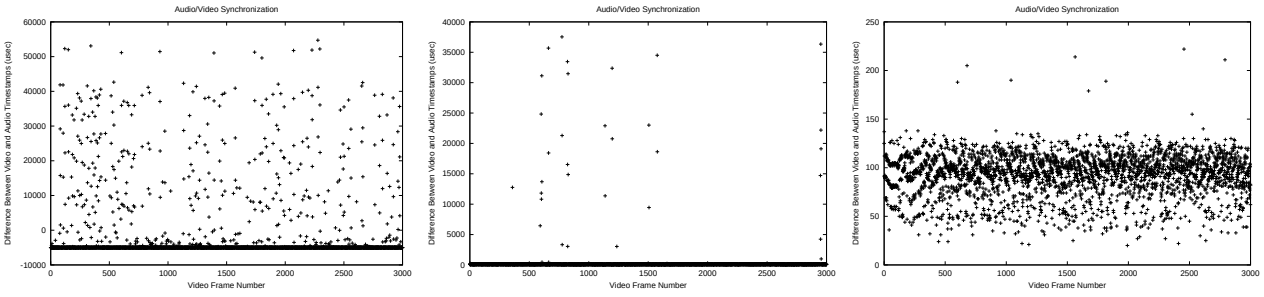
Non-kernel CPU load Figure 2 shows the audio/video skew in mplayer on a Linux and a time-sensitive Linux kernel when a CPU stress test is the competing load. This competing load runs an infinite loop consuming as much CPU as possible. Figure 2(a) shows that for standard Linux the maximum skew is large and close to 50 ms when the X11 server is run at a non-real time priority. For most data points, the skew lies between -5 ms to 5 ms because of Linux's 10 ms timer resolution. Figure 2(b) shows that the skew for time-sensitive Linux, when X11 is run at a non-real time priority, is still large (up to $35ms$) but does not show oscillations between -5 ms and 5 ms. Finally, Figure 2(c) shows that the skew for time-sensitive Linux improves considerably and is less than 250 us when the X11 server runs at real-time priority. The real-time priority value of X11 is the same as the priority assigned to mplayer.

These figures show that time-sensitive Linux works well on a non-kernel CPU load as long as the HLP protocol, described in Section 3.3.2, is used to assign priorities to time-sensitive tasks and to server tasks with the shared resources. Linux with the X11 server at real-time priority still has a skew between -5 ms to 5 ms because of the timer resolution.

As a result of this experiment, the rest of the experiments in this section are run with mplayer and X11 at real-time priority to avoid any user-level priority inversion effects.

Kernel CPU Load The second experiment compares the audio/video skew in mplayer between Linux and time-sensitive Linux when the background load copies a large 8 MB memory buffer to a file with a single `write` system call. Figure 3(a) shows the audio/video skew is as large as 90 ms for Linux. In this case, the kernel moves the data from the user to the kernel space in a non-preemptible section. Figure 3(b) shows that the maximum skew is less than 400 us for time-sensitive Linux. This improvement occurs as a result of improved kernel preemptibility for large write calls in time-sensitive Linux.

File System Load The third experiment compares the audio/video skew in mplayer between Linux and time-sensitive Linux when the background load repeatedly copies a compiled Linux kernel sources directory recursively and then flushes the file system. This directory has 13000 files and 180 MB of data and is stored on the Linux `ext2` file system. The kernel uses DMA for transferring disk data. Figure 4(a) shows that the skew under Linux can be as high as 120 ms while Figure 4(b) shows that skew is less than 200 us under time-sensitive Linux. This result shows that time-sensitive Linux can provide low latencies even under heavy file-system and disk load.

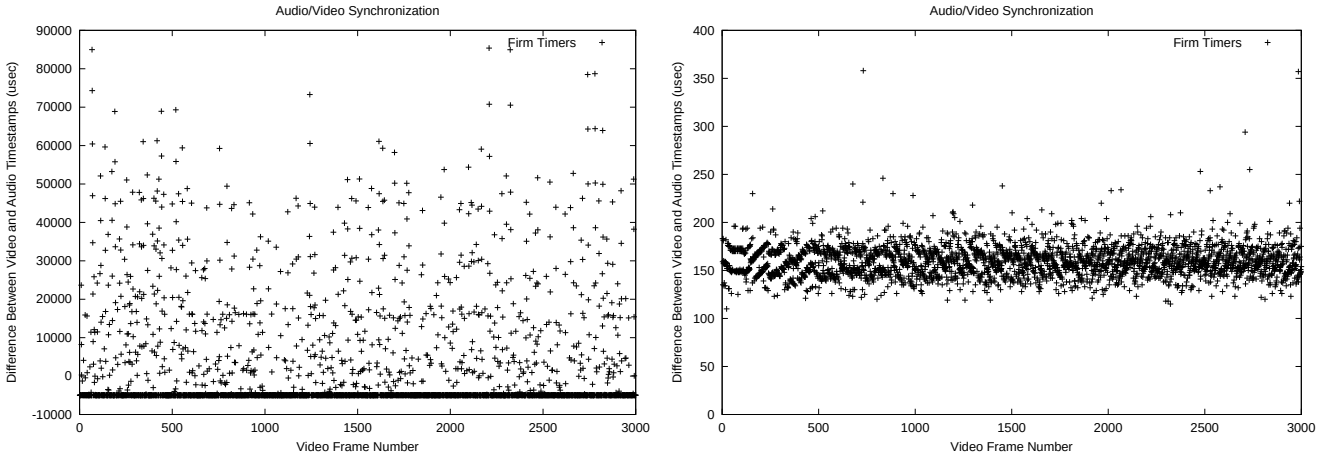


(a) Linux, X server non-real time

(b) Time-sensitive Linux, X server non-real time

(c) Time-sensitive Linux, X server real-time

Figure 2: Audio/Video Skew on Linux and time-sensitive Linux. Background load is a CPU stress test that run an empty loop. Note that the three figures have different scales, and that the maximum skew in Figure (c) is much smaller than the maximum skew in the other two cases.



(a) Linux

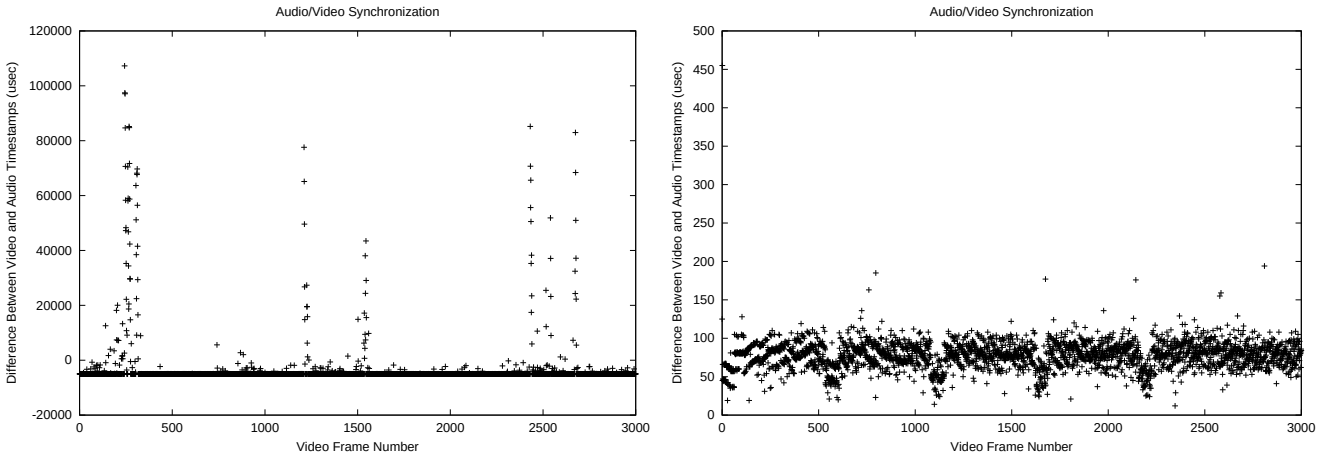
(b) Time-sensitive Linux

Figure 3: Audio/Video Skew on Linux and time-sensitive Linux. Background load copies a 8 MB buffer from user level to a file with a single `write` call. Note that the two figures have different scales, and that the maximum skew in Figure (b) is much smaller than in Figure (a)

4.2.2 Proportion-Period Scheduler

Our original motivation for implementing a time-sensitive Linux system was to accurately implement a proportion-period scheduler. This scheduler is used to provide a reservation mechanism for a higher-level feedback-based real-rate scheduler[22]. The real-rate scheduler uses an application-specific progress rate metric in time-sensitive tasks to automatically assign correct allocations to such tasks. For example, the progress of a producer or consumer of a bounded buffer can be inferred by measuring the fill-level of the bounded buffer. If the buffer is full, the consumer is falling behind and needs more resources while the producer needs to be slowed down.

The feedback allocation accuracy depends (among other factors) on the accuracy of actuating proportions. There are three sources of inaccuracy in our proportion-period scheduler implementation on Standard Linux: 1) the period boundaries are quantized to multiples of the timer resolution or 10 ms, 2) the policing of proportions is also limited to the same value because timers have to



(a) Linux

(b) Time-sensitive Linux

Figure 4: Audio/Video Skew on Linux and time-sensitive Linux. Background load repeatedly copies a compiled Linux kernel sources directory recursively and then flushes the file system. Note that the two figures have different scales, and that the maximum skew in Figure (b) is much smaller than in Figure (a).

be used to implement policing, and 3) heavy loads cause long non-preemptible paths and thus large jitter in period boundaries and proportion policing. These inaccuracies introduce noise in the system that can cause large allocation fluctuations even when the input progress signal can be captured perfectly and the controller is well-tuned.

Our time-sensitive Linux's proportion-period scheduler uses firm-timers for implementing period boundaries and proportion policing. To evaluate the accuracy of this scheduler, we ran two processes with proportions of 40% and 20% and periods of 8192 us and 512 us respectively.[3] These processes were run first on an unloaded system to verify the correctness of the scheduler. Then, we evaluated the scheduler behavior when the same processes were run with competing file system load (described in Section 4.2.1). In this experiment each process runs a tight loop that repeatedly invokes the gettimeofday system call to measure the current time and stores this value in an array. The scheduler behavior is inferred at the user-level by simply measuring the time difference between successive elements of the array. A similar technique is used by Hourglass [18].

Table 1 shows the maximum deviation in the proportion allocated and the period boundary for each of the two processes. This table shows that the proportion-period scheduler allocates resources with a very low deviation of less than $25us$ on a lightly loaded system. Under high file system load the results show larger deviations. These deviations occur because execution time is "stolen" by the kernel interrupt handling code which runs at a higher priority than user-level processes in Linux.

One way to improve the proportion-period scheduling performance in the presence of heavy file system load is to defer certain parts of interrupt processing after real-time processes. We are currently investigating this solution.

An alternative method for evaluating the scheduler behavior is to use a kernel tracer, such as LTT[23], that can register the occurrence of certain key events in the kernel. These events can be analyzed later after program execution. Kernel tracers are often used in real-time systems for verifying the temporal correctness of the kernel and of time-sensitive applications. We ported LTT to time-sensitive Linux, and Figure 5 shows a sample session analyzing the schedule generated in the previous experiments. The trace visualizer application shows the schedule, visualizing processes execution (an executing process is shown in black): the two proportion-period processes, marked as "unnamed child" by the trace visualizer and characterized by PIDs 2493 and 2492, are easily recognizable, because they use most of the CPU

[3]The current proportion-period scheduler allows task periods that are multiples of 512 us. While this period alignment restriction is not needed for a proportion-period scheduler, it simplifies feedback-based adjustment of task proportions.

	No Load		File System Load	
	Max Proportion Deviation	Max Period Deviation	Max Proportion Deviation	Max Period Deviation
Task 1 Proportion: 40%, 3276.8 us Period: 8192 us	0.3% (≃25us)	5us	6% (≃490us)	534us
Task 2 Proportion: 20%, 102.4 us Period: 512 us	0.7% (≃3us)	10us	4% (≃20us)	97us

Table 1: Deviation in proportion and period for two processes running on the proportion-period scheduler on time-sensitive Linux.

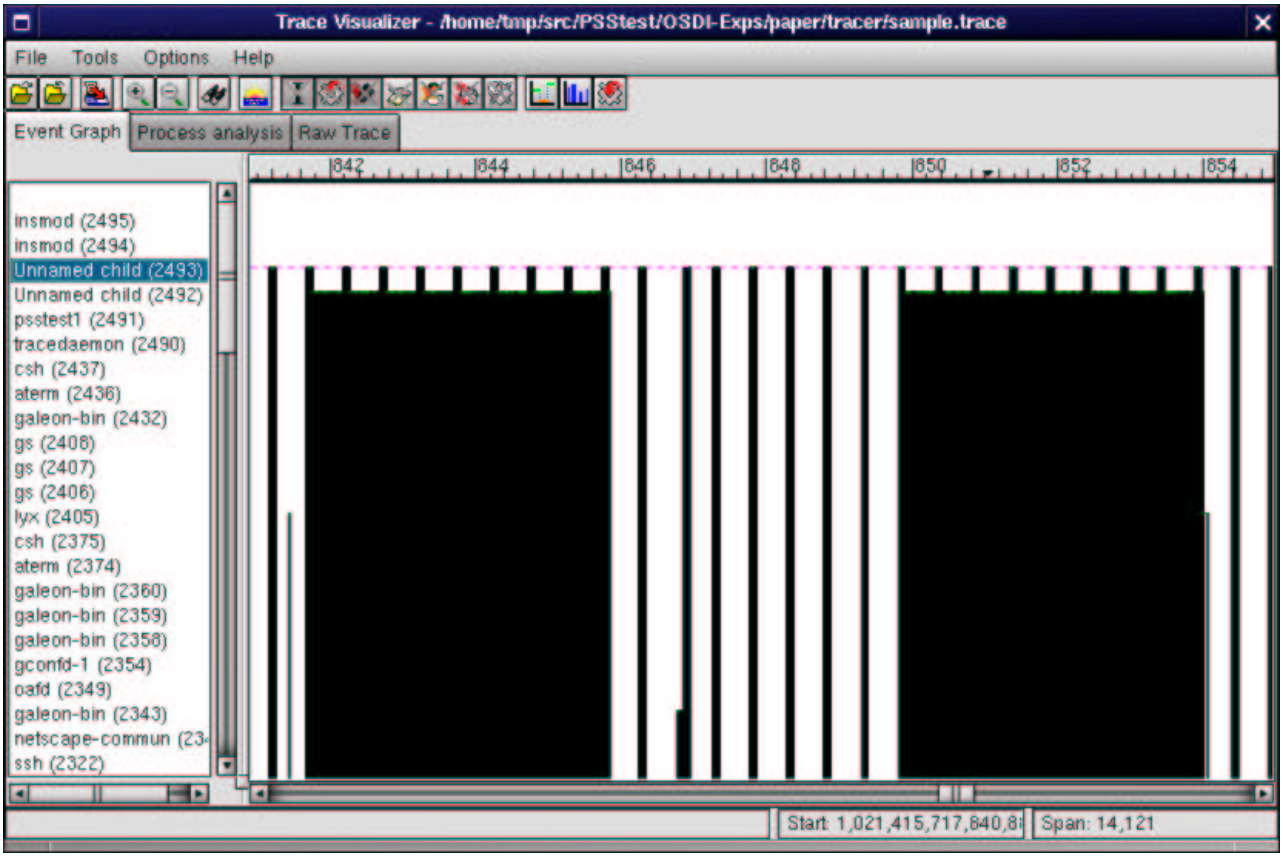


Figure 5: Linux kernel tracer (LTT) helps to visualize the schedule generated by two proportion-period processes.

time. Also note that their execution is regular, and coincides with the one expected for two processes with proportion-periods $(40\%, 8192)$ and $(20, 512)$. All the non time-sensitive processes can execute when processes 2492 and 2493 exhausted their proportions: for example, a small execution of the lyx editor is visible after the first period of process 2493.

4.3 System Overhead

High resolution timers in a general purpose OS can potentially have high overheads. To mitigate this overhead, our firm timers implementation combines hard and soft timers. In this section, we present experiments to highlight the advantages of firm timers as compared to pure hard timers and show that the firm-timer overhead is acceptable.

In all these experiments, we measure the performance

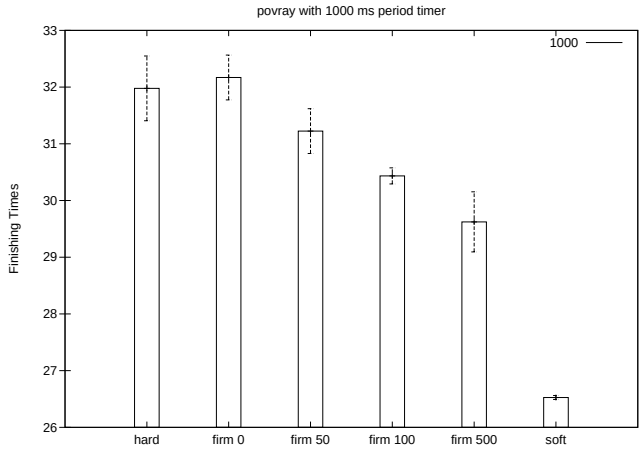


Figure 7: Comparison between Hard and Firm timers with different accuracy.

of a throughput-oriented application when one or more time-sensitive processes are run in the background to stress the firm timers mechanism. For the throughput application, we selected `povray`, a ray-tracing application and used it to render a standard benchmark image called `skyvase`.

Figure 6 shows the performance overhead of firm timers as compared to standard Linux timers when multiple periodic time-sensitive processes are running simultaneously. We conducted experiments with 20 and 50 timers running with 10 ms and 100 ms period. The overhead is defined as the ratio of the time needed by `povray` to render the image in time-sensitive Linux and the time needed to render the same image in standard Linux. The figures show the overhead for time-sensitive Linux with pure hard timers, firm timers with accuracy 500 us, and pure soft timers. These figures show that pure soft timers do not have any overhead as compared to standard Linux timers (except when 50 time-sensitive processes with period 10 ms are run), hard timers have a slightly greater overhead and firm timers, in this case, do not provide much improvement.

We also performed the same experiment but with periodic processes running at higher frequencies. Figure 7 shows the time needed to render the image when 20 periodic processes are run with a period of 1 ms. We do not compare these results with Linux because Linux does not support 1 ms timer accuracy. The benefit of the firm timers mechanism for improving throughput becomes obvious when the process periods are made shorter.[4]

[4]Note that the low execution time with pure soft timers is because not all timers fired at the correct times. Thus this value should not be considered in the comparison.

The previous experiments show that pure hard timers have lower overhead in some cases and firm timers have lower overhead in other cases. This result can be explained by the fact that there is a cost associated with checking whether a soft timer has expired. Thus, the soft timers mechanism is effective in reducing overhead when enough of these checks result in the firing of a soft timer. Otherwise the firm-timer overhead as compared to pure hard timers will be higher.

More formally, the previous behavior can be explained as follows. Let T be the total number of timers that must fire in a given interval of time, H the number of hard timers that fire, S be the number of soft timers that fire (hence, $T = H + S$) and C be the number of checks for soft timers expirations. Let C_h be the cost for firing a hard timer, C_s be the cost of firing a soft timer, and C_c be the cost of checking if some soft timer has expired. The total cost of firing firm timers is $C_cC + C_hH + C_sS$. If pure hard timers are used then the cost is C_hT. Hence, firm timers reduce the system overhead if

$$C_cC + C_hH + C_sS < C_hT \Rightarrow$$

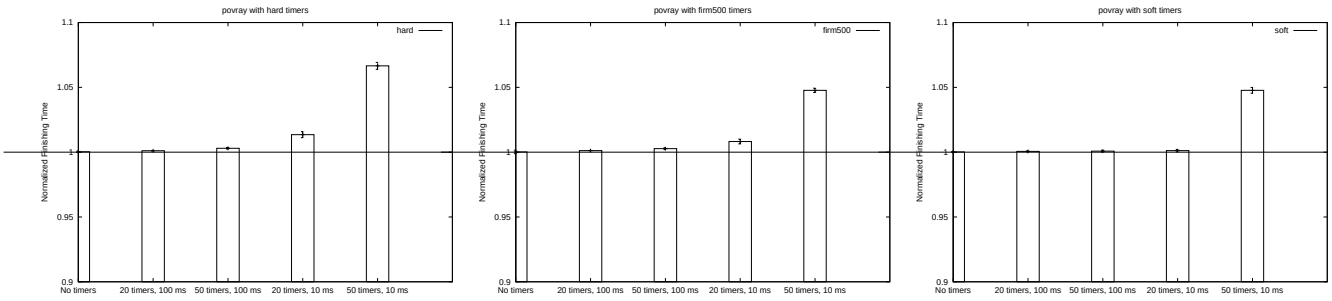
$$C_cC < C_h(T - H) - C_sS \Rightarrow$$

$$C_cC < (C_h - C_s)S \Rightarrow$$

$$S/C > C_c/(C_h - C_s)$$

Thus when the ratio of the number of the soft timers that fire to the number of soft timer checks is sufficiently large (i.e., it is larger than $C_c/(C_h - C_s)$), then firm timers are effective in reducing the high-resolution timers overhead. In our experiments, we measured $C_h = 8us$, $C_s = 1us$, and $C_c = 0.15177us$, hence the firm timers mechanism becomes effective when $S/C > 0.15177/(8 - 1) = 0.021682$, that is to say when more than 2.1% of the soft timer checks result in a soft timer to be fired.

Note that the number of checks C depends on the amount of interrupts and system calls that happen in the machine, whereas the amount of soft timers that fire S depends on how the checks and the timers' deadlines are distributed in time. The original work on soft timers [5] studied these distributions for a number of workloads. Their results show that for many workloads the distributions are such that checks often occur close to deadlines (thus increasing S/C), although how close is very workload dependent. Firm timers have the benefit of assuring



(a) Hard Timers (b) Firm Timers (accuracy $500us$) (c) Soft Timers

Figure 6: Overhead of firm timers in time-sensitive Linux as compared to standard Linux.

response even for workloads with poor distributions, yet retaining the performance benefits of soft timers when the workload permits.

5 Conclusions

This paper describes the design and implementation of a time-sensitive Linux system that can support applications requiring fine-grained resource allocation and low-latency response. The three key techniques that we have investigated in the context of our time-sensitive Linux system are firm timers for accurate timing, kernel preemptibility for improving kernel responsiveness and proportion-period scheduling for providing precise allocations to tasks. Our experiments show that integrating these techniques helps provide allocations to time-sensitive tasks with a variation of less than 400 us even under heavy CPU, disk and file system load. We show that the overhead of time-sensitive Linux on throughput-oriented applications is low and thus such a system can be used effectively for time-sensitive and general-purpose applications.

Although the first results presented in this paper are promising, TSL still need further investigation, since there are open issues related, for example, to interrupt service, to network latencies, and to firm timers performance. For firm timers in particular, we interested in investigating whether real workloads commonly lead to the $S/C > K$ condition under which firm timers are effective.

References

[1] Montavista software - powering the embedded revolution. `http://www.mvista.com/`.

[2] Mplayer - movie player for linux. `http://www.mplayerhq.hu`.

[3] The X window system. www.x.org.

[4] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. A Measurement-Based Analysis of the Real-Time Performance of the Linux Kernel. In submission to the Real Time Technology and Applications Symposium (RTAS), March 2002.

[5] Mohin Aron and Peter Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Systems*, August 2000.

[6] Michael Barabanov and Victor Yodaiken. Real-time linux. *Linux Journal*, March 1996.

[7] Randy Brown. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *CACM*, 31(10):1220–1227, October 1988.

[8] Intel Corporation, editor. *Pentium Pro Family Developer's Manual*, chapter 7.4.15. Intel, December 1995.

[9] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Rosu, and Marcel-Catalin Rosu. An overview of the rialto real-time

architecture. In *In Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

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

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

[12] Robert Love. The Linux Kernel Preemption Project. `http://kpreempt.sourceforge.net`.

[13] P. Mantegazza, E. Bianchi, L. Dozio, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, 72, 2000.

[14] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994. To appear. (This is a condensed version of tech report CMU-CS-93-157.).

[15] Clifford W. Mercer and Hideyuki Tokuda. Preemptibility in real-time operating systems. In *In Proceedings of the 13th IEEE Real-Time Systems Symposium*, December 1992.

[16] Andrew Morton. Linux scheduling latency. `http://www.zip.com.au/~akpm/linux/schedlat.html`.

[17] Shui Oikawa and Raj Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the IEEE Real-Time Systems Symposium Work-In-Progress*, Madrid, December 1998.

[18] John Regehr. Inferring scheduling behavior with hourglass. In *Proceedings of the Freenix Track of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[19] S. Savage and H. Tokuda. Rt-mach timers: Exporting time to the user. In *In Proceedings of USENIX 3rd Mach Symposium*, April 1993.

[20] Lui Sha, Raghunathan Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1184, September 1990.

[21] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1998.

[22] David Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*. USENIX, February 1999.

[23] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX Annual Conference*, San Diego, CA, June 2000.

[24] Yu-Chung and Kwei-Jay Lin. Enhancing the Real-Time Capability of the Linux Kernel. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hiroshima, Japan, October 1998.