Supporting Time-Sensitive Applications on a Commodity OS

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

Abstract

Commodity operating systems are increasingly being used for serving time-sensitive applications. These applications require low-latency response from the kernel and from other system-level services. In this paper, we explore various operating systems techniques needed to support time-sensitive applications and describe the design of our Time-Sensitive Linux (TSL) system. We show that the combination of a high-precision timing facility, a well-designed preemptible kernel and the use of appropriate scheduling techniques is 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 satisfy the constraints of time-sensitive applications in a commodity operating system without significantly 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 (e.g., soft modems [8]) or quick response to external events such as frame capture (e.g., video conferencing).

To support these time-sensitive applications, a general-purpose operating system (OS) must respect the application's timing constraints. To do so, resources must be allocated to the application at the appropriate times. This paper shows that there are three important requirements for achieving timely resource allocation: a high-precision timing facility, a well-designed preemptible kernel and the use of appropriate scheduling techniques. Each of these requirements have been addressed in the past with specific mechanisms, but unfortunately 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 three types of timers, one-shot timers available on modern hardware [7], soft timers [2], and periodic timers to provide accurate timing with low overhead. Second, we use fine-grained kernel preemptibility to obtain 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, commodity 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 commodity system such as Linux [17, 22]. These approaches include improved kernel preemptibility and a more generic scheduling framework. However, since their focus is hard real-time, they do not evaluate the performance of non-real time applications.

In contrast, TSL focuses on integrating efficient support for time-sensitive applications in a commodity OS without significantly degrading the performance of traditional applications. Hence, one of the main contributions 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

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

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 commodity systems. Section 3 describes the techniques that we have used to implement TSL. Section 4 evaluates the behavior of several time-sensitive applications and presents overheads of TSL. Finally, in Section 5, we state our conclusions.

1.1 Related Work

The scheduling problem has been extensively studied by the real-time community [10, 19]. 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 [22] provides a generic real-time scheduling framework. These kernels tackle the practical systems issues mentioned above with techniques similar to the techniques presented in this paper. For example, kernel preemptibility is used by Timesys Linux and MontaVista Linux [12]. However, while these kernels work well for time-sensitive applications, their performance overhead on throughput-oriented applications is not clear.

The SMaRT [16] and Linux-SRT [6] kernels share our goal of supporting time-sensitive applications on commodity OSs. Before implementing SMaRT, Nieh, et al. [15] showed that tuning the time-sharing or the real-time priorities of applications required a great deal of experimentation in SVR4 Unix and often lead to unpredictable results. They showed empirically, similar to our priority-assignment protocol, that assigning a higher real-time priority to the X server improved the performance of video. Later, Nieh implemented the SMaRT real-time scheduler on Solaris that dynamically balances between the needs of different jobs by giving proportional shares to each job and a bias that improves the responsiveness of interactive and real-time jobs. This scheduler complements our approach of using priorities or proportion-period scheduling, as appropriate, for time-sensitive tasks.

Linux-SRT supports multiple real-time scheduling policies and implements a timing mechanism that is more accurate than standard Linux. However, it doesn't incorporate kernel preemption or discuss the issue of time-sensitive performance under heavy system load. Linux-SRT recognizes that the timing behavior of an application depends on shared system services such as the X server and thus modifies the X server to prioritize graphics rendering based on the scheduling parameters of tasks. We use a simpler priority-assignment protocol to achieve the same effect without requiring any modifications to the X server.

A different approach for providing real-time performance is used by systems such as RTLinux [4], which decrease timing unpredictability in the system by running Linux as a background process over a small real-time executive. This solution provides good real-time performance, but does not provide it to Linux applications. Also, native real-time threads use a different and less evolved application binary interface (ABI) as compared to the Linux interface 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 [18] and has subsequently been used by several other systems [17]. 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 fine-grained timing mechanism and can be mitigated by the soft-timer mechanism [2]. Thus, our firm-timer implementation uses soft timers.

Finally, the Nemesis operating system [9] is designed for multimedia and other time-sensitive applications. However, its structure and API is very different from the standard programming environment provided by 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 commodity environment.

2 Time-Sensitive Requirements

To satisfy a time-sensitive application's temporal constraints, resources must be allocated at "appropriate" times. These appropriate times are determined by events of interest to the application, such as readiness of a video frame for display. In response to these events, the application is scheduled or activated. Hence, we view the time-line of the application as a sequence of such *events* and the corresponding *activation*s. For example, Fig-

ure 1 shows an event and its activation. It depicts the execution sequence in a system after a wall-clock time event. We call the latency between the event and its activation, *kernel latency.* Kernel latency should be low in a time-sensitive system. As the figure shows, kernel latency has three components that we call *timer latency*, *preemption latency* and *scheduling latency*. To reduce these components of kernel latency, there are three requirements: 1) an accurate timing mechanism, 2) a responsive kernel and 3) an appropriate CPU scheduling algorithm. These requirements are described below.

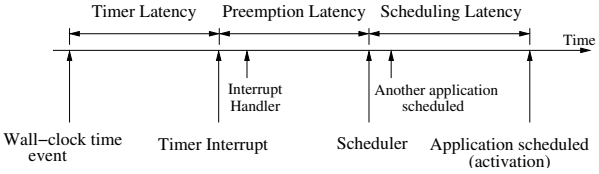


Figure 1: Execution sequence after wall-clock time expiration event.

Timing Mechanism: An accurate timing mechanism is crucial for reducing latency since timer resolution is the largest source of latency in an operating system such as Linux [1]. Such a mechanism can be implemented efficiently by using two techniques, one-shot timers available on most common modern hardware and soft timers [2]. 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 cheaply in a few CPU cycles.

Soft timers check and run expired timers at strategic points in the kernel and help in 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 small.

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 for improving kernel response 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. For example, soft modems, which use the main processor to execute modem functions, require computational processing at least every 16 ms or else modem audio and other multimedia audio applications can get impaired [8]. Then, we allocate a fixed proportion of the CPU every delay period 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 [2] 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 [13] 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 [12]. Real-time operating systems integrate these solutions for time-sensitive tasks but tend to ignore the performance overhead of their 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 X server for graphical display [21] on a Linux system, has the same components of kernel latency described above. In fact, the simple scheduling models presented above assume that tasks are independent. In a real application, tasks can be interdependent which can cause priority inversion problems [15]. For example, in Section 4.2.1 we show that a multimedia player that uses the X server for display can perform sub-optimally due to priority inversion, even if the kernel allocates resources correctly. The X 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 Time-Sensitive Linux (TSL) that provides accurate timers, a responsive kernel, and time-sensitive scheduling algorithms to support the requirements highlighted above.

3 Implementing Time-Sensitive Linux

We propose three specific techniques, firm timers, fine-grained kernel preemptibility and proportion-period CPU scheduling for reducing the three components of kernel latency. We have integrated these techniques in the Linux kernel to implement TSL.

3.1 Firm Timers

Firm timers provide an accurate timing mechanism with low overhead by exploiting the benefits associated with three different approaches for implementing timers: one-shot timers, soft timers and periodic timers.

Traditionally, commodity operating systems have implemented their timing mechanism with periodic timers. These timers are normally implemented with periodic timer interrupts. For example, on Intel x86 machines, these interrupts are generated by the Programmable Interval Timer (PIT), and on Linux, the period of these interrupts is 10 ms. As a result, the maximum timer latency is 10 ms. This latency can be reduced by reducing the period of the timer interrupt but this solution increases system overhead because the timer interrupts are generated more frequently.

To reduce the overhead of timers, it is necessary to move from a periodic timer interrupt model to a one-shot timer interrupt model where interrupts are generated only when needed. Consider two tasks with periods 5 and 7 ms. With periodic timers and a period of 1 ms, the maximum timer latency would be 1 ms. In addition, in 35 ms, 35 interrupts would be generated. With one-shot timers, interrupts will be generated at 5 ms, 7 ms, 10 ms, etc., and the total number of interrupts in 35 ms is 11. Also, the timer latency will be close to the interrupt service time, which is relatively small. Hence, one-shot timers avoid unnecessary interrupts and reduce timer latency.

3.1.1 Firm Timers Design

Firm timers, at their core, use one-shot timers for efficient and accurate timing. One-shot timers generate a timer interrupt at the next timer expiry. At this time, expired timers are dispatched and then finally the timer interrupt is reprogrammed for the next timer expiry. Hence, there are two main costs associated with one-shot timers, timer reprogramming and fielding timer interrupts. Unlike periodic timers, one-shot timers have to be continuously reprogrammed for each timer event. More importantly, as the frequency of timer events increases, the interrupt handling overhead grows until it limits timer frequency. To overcome these challenges, firm timers use inexpensive reprogramming available on modern hardware and combine soft timers (originally proposed by Aron and Druschel [2]) with one-shot timers to reduce the number of hardware generated timer interrupts. Below, we discuss these points in more detail.

While timer reprogramming on traditional hardware has been expensive (and has thus encouraged using periodic timers), it has become inexpensive on modern hardware such as Intel Pentium II and later machines. For ex-

ample, reprogramming the standard programmable interval timer (PIT) on Intel x86 is expensive because it requires several slow `out` instructions on the ISA bus. In contrast, our firm-timers implementation uses the APIC one-shot timer present in newer Intel Pentium class machines. This timer resides on-chip and can be reprogrammed in a few cycles without any noticeable performance penalty.

Since timer reprogramming is inexpensive, the key overhead for the one-shot timing mechanism in firm timers lies in fielding interrupts. Interrupts are asynchronous events that cause an uncontrolled context switch and result in cache pollution. To avoid interrupts, firm timers use soft timers, which poll for expired timers at strategic points in the kernel such as at system call, interrupt, and exception return paths. At these points, the working set in the cache is likely to be replaced anyway and hence polling and dispatching timers does not cause significant additional overhead. In essence, soft timers allow voluntary switching of context at "convenient" moments.

While soft timers reduce the costs associated with interrupt handling, they introduce two new problems. First, there is a cost in polling or checking for timers at each soft-timer point. Later, in Section 4.3.2, we analyze this cost in detail and show that it can be amortized if a certain percentage of checks result in the firing of timers. Second, this polling approach introduces timer latency when the checks occur infrequently or the distribution of the checks and the timer deadlines are not well matched.

Firm timers avoid the second problem by combining one-shot timers with soft timers by exposing a system-wide *timer overshoot* parameter. With this parameter, the one-shot timer is programmed to fire an overshoot amount of time after the next timer expiry (instead of exactly at the next timer expiry). In some cases, an interrupt, system call, or exception may happen after a timer has expired but before the one-shot APIC timer generates an interrupt. At this point, the timer expiration is handled and the one-shot APIC timer is again reprogrammed an overshoot amount of time after the next timer expiry event. When soft-timers are effective, firm timers repeatedly reprogram the one-shot timer for the next timer expiry but do not incur the overhead associated with fielding interrupts.

The timer overshoot parameter allows making a trade-off between accuracy and overhead. A small value of timer overshoot provides high timer resolution but increases overhead since the soft timing component of firm timers are less likely to be effective. Conversely, a large value decreases timer overhead at the cost of increased maximum timer latency. The overshoot value can be changed dynamically. With a zero value, we obtain one-shot timers (or hard timers) and with a large value, we obtain soft timers. A choice in between leads to our hybrid firm timers approach. This choice depends on the timing accuracy needed by applications.

3.1.2 Firm Timers Implementation

Firm timers in TSL maintain a timer queue for each processor. The timer queue is kept sorted by timer expiry. The one-shot APIC timer is programmed to generate an interrupt at the next timer expiry event. 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 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 available on a modern machine, a one-shot timer has a theoretical accuracy of 10 nanoseconds. However, in practice, the time needed to field timer interrupts is significantly higher and is the limiting factor for timer accuracy.

Soft timers are enabled by using a non-zero timer overshoot value, in which case, the APIC timer is set an overshoot amount after the next timer event. Our current implementation uses a single global overshoot value. It is possible to extend this implementation so that each timer or an application using this timer can specify its desired overshoot or timing accuracy. In this case, only applications with tighter timing constraints cause the additional interrupt cost of more precise timers. The overhead in this alternate implementation involves keeping an additional timer queue sorted by the timer expiry plus overshoot value.

The data structures for one-shot timers are less efficient than for periodic timers. For instance, periodic timers can be implemented using calendar queues [5] 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 period of the timer interrupt. Calendar queues need this fixed 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.

To derive the data structure efficiency benefits of periodic timers, firm timers combine the periodic timing mechanism with the one-shot timing mechanism for timers that need a timeout longer than the period of the periodic timer interrupt. 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. Consequently, 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 becomes more efficient. Note that operating systems generally perform periodic activity such as time keeping, accounting and profiling at each periodic tick interrupt and thus the dual wakeup does not add any additional cost. An additional benefit of this approach is that timer drift in firm timers is limited to a small fraction of the period of the periodic timer interrupt assuming, as in Linux, that the periodic interrupt is synchronized to the global NTP protocol while the APIC timer is not.

The firm timer expiration times are specified as CPU clock cycle values. In an x86 processor, the current time in CPU cycles in stored in a 64 bit register. Timer expiration values can be stored as 64 bit quantities also but this choice involves expensive 64 bit time conversions from CPU cycles to memory cycles needed for programming the APIC timer. A more efficient alternative for time conversion is to store the expiration times as 32 bit quantities. However, this approach leads to quick roll over on modern CPUs. For example, on a two GHz processor, 32 bits roll over every second. Fortunately, firm timers are still able to use 32 bit expiration times because they use periodic timers for long timeouts and use one-shot timer expiration values only within a periodic tick.

We want to provide the benefits of the firm timer 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 in TSL to use firm timers without changing the interface of these calls. As a result, unmodified applications automatically get increased timer accuracy in our system as shown in Section 4.2.

3.2 Fine-Grained Kernel Preemptibility

A kernel is responsive when its non-preemptible sections, which keep the scheduler from being invoked to schedule a task, are small. There are two main reasons why the scheduler may not be able to run. One is that interrupts might be disabled. For example, if the timer interrupt in Figure 1 is disabled, the timer task can only enter the ready queue when the interrupt is re-enabled. Another, potentially more significant reason is that another thread may be executing in a critical section in the kernel. For example, the timer task upon entering the ready queue will be scheduled only when the other thread exits its non-preemptible critical 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 commodity kernels disable preemption for the entire period of time when a thread is in the kernel, i.e., when an interrupt fires or for the duration of a system call, except before certain well-known long operations are invoked by the kernel. For example, they generally allow preemption before invoking disk I/O operations. Unfortunately, with this structure, preemption latency under standard Linux can be greater than 30 ms [1].

One approach that reduces preemption 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 the scheduler when it reaches these preemption points. 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 [13]. Preemption latency in such a kernel decreases to the maximum time between two preemption points.

Another approach, used in most real-time systems, is to allow preemption anytime the kernel is not accessing shared data structures. To support this fine level of kernel preemptibility, shared kernel data must be explicitly protected using mutexes or spinlocks. The Linux preemptible kernel project [11] uses this approach and disables kernel preemption only when a spinlock is held or an interrupt handler is executing to protect the consistency of shared kernel data structures. In a preemptible kernel, preemption latency is determined by the maximum amount of time for which a spinlock is held inside the kernel and the maximum time taken by interrupt service routines.

The preemptible kernel approach can have high preemption latency when spinlocks are held for a long time. To address this problem, a third approach combines explicit preemption with the preemptible kernel approach by releasing (and reacquiring) spin-locks at strategic points in long code sections that access shared data

structures within spinlocks. This approach is used by Robert Love's lock-breaking preemptible kernel patch for Linux [11].

Our previous evaluation [1] shows that these approaches work fairly well for reducing preemption latency and should be incorporated in the design of any responsive kernel. As expected, the combined preemption approach is slightly superior to the first two approaches and, consequently, we have incorporated Robert Love's lock-breaking preemptible kernel patch in TSL for improving kernel responsiveness. Our experiments with real applications on TSL 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 scheduling latency. We use a combination of the proportion-period and priority models, described in 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 and provides compatibility with applications based on a POSIX scheduler, 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 commodity time-sensitive 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 proportion-period allocation model automatically provides temporal protection because each task is allocated a fixed proportion of the CPU at each task period. The period of a task is related to some application-level delay requirement of the application, such as the period or the jitter requirements of a 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 [10, 19].

We implemented a proportion-period CPU scheduler in Linux by using real-time scheduling techniques (EDF priority assignment) and by allowing a task to execute as a real-time task for a time Q every period T. The reserved time Q is equal to the product of the task's proportion P and its period T, and the end of each period is used as a deadline. After executing for a time Q, the task is blocked (or scheduled as a non real-time task) until its next period. When two tasks have the same deadline, the one with the smallest remaining capacity is scheduled to reduce the average finishing time.

The proportion-period model and its variants have been implemented and used extensively in the past [9]. While several of these schedulers are more sophisticated that ours, the main focus of this paper is to show that these schedulers can be implemented accurately in TSL and hence can improve the accuracy of scheduling analysis.

3.3.2 Priority CPU Scheduling

In the priority model, real-time priorities are assigned to time-sensitive tasks based on application needs [10]. Since the fixed priority model does not provide temporal protection, TSL schedules fixed priority tasks in the background with respect to proportion-period tasks. The only exception to this rule is with shared server tasks because they can cause *priority inversion*, and we must use a proper resource sharing protocol to bound its effects. In general, priority inversion occurs when an application is composed of multiple tasks that are interdependent. For example, consider a simple example of a video application consisting of a client and an X 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 X 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 X 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 [19] 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 X server must have the

highest priority among all time-sensitive clients accessing it. In this way, the X server cannot be 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.

3.3.3 TSL Scheduling Model

If fixed priority tasks are the only ones accessing the X server, then the server can be scheduled with the maximum fixed priority, but in the background with respect to the proportion-period tasks. If, on the other hand, proportion-period tasks require access to the X server, then it must be scheduled with the highest priority in the system. This is the exception to the rule of scheduling fixed priority tasks in the background with respect to proportion-period tasks. Due of this exception, the shared server can jeopardize the proportion-period guarantee. Hence, the server must be "trusted", i.e., its execution time must be known or at least bounded.

Using real-time terminology, the shared server causes blocking time on proportion-period tasks. If this blocking time is known, it can be accounted in the guarantee of proportion-period tasks [19, 3]. Otherwise, the only safe thing to do is to leave some "unreserved CPU time". Hence, the admission test for proportion-period tasks is not $\sum_i P_i \leq 1$, but $\sum_i P_i \leq U^{max} < 1$. We have found that $U^{max} = 0.9$ is a reasonable value and works well in practice.[1]

4 Evaluation

This section describes the results of experiments we performed to evaluate 1) the behavior of time-sensitive applications running on TSL, and 2) the overheads of TSL. In these experiments TSL is derived from Linux version 2.4.16. It incorporates our firm timers, Robert Love's lock-breaking preemptible kernel patch and our proportion-period scheduler. 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.

[1] Note that if the blocking times are unknown it is impossible to provide a hard guarantee.

4.1 Micro Benchmarks

Before evaluating the impact of the latency reduction techniques used in TSL on real applications, we performed micro-benchmarks for evaluating the components of kernel latency, as described in Section 2. These components consist of timer latency, preemption latency and scheduling latency. We evaluated the first two components in isolation by running a time-sensitive process that needs to sleep for a specified amount of time (using the `nanosleep()` system call) and measures the time that it actually sleeps. In our first set of experiments, we evaluated timer latency and showed that it is 10 ms in standard Linux while firm timers reduce it to a few microseconds on TSL.

Next, we evaluated preemption latency when a number of different system loads are run in the background. We compared preemption latency under Linux, Andrew Morton's Linux with explicit preemption [13] and Robert Love's preemptible kernel and lock-breaking preemptible kernels [11]. The first interesting result was that on standard Linux the worst case preemption latency can be larger than 100 ms (when the kernel copies large amounts of data between kernel and user space) but in the common case preemption latency is less than 10 ms and is generally hidden by timer latency. However, when firm timers are used, preemption latency becomes more visible, and it is easy for latencies to be larger than 5 ms. Using the explicit preemption and kernel preemptibility techniques described in Section 3.2, preemption latency can be greatly reduced, and thus TSL provides a maximum kernel latency of less than 1 ms on our test machine even when the system is heavily loaded. The full details of these experiments and more results are presented in our previous paper [1], which we have briefly summarized here for the reader's convenience.

4.2 Latency in Real Applications

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

ric for mplayer. The latency metric for the proportion-period scheduler is maximum error in the allocation and period boundary.

4.2.1 Mplayer

Mplayer [14] is an audio/video player that can handle several different media formats. Mplayer synchronizes audio and video streams by using time-stamps that are associated with the audio and video frames. The audio card is used as a timing source and when a video frame is decoded, its time-stamp is compared with the time-stamp of the currently playing audio sample. If the video time-stamp is smaller than the audio time-stamp then the video frame is late and the video is immediately displayed. Otherwise, the system sleeps until the time difference between the video and audio time-stamps and then displays the video.

On a responsive kernel with sufficient available CPU capacity, audio/video synchronization can be achieved by simply sleeping for the correct amount of time. Unfortunately, if the kernel is unresponsive or has a coarse timing mechanism, 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 hence this paper only presents results for audio/video synchronization skew.

We compare the audio/video skew of mplayer between standard Linux and TSL 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 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 100 seconds at real-time priority.

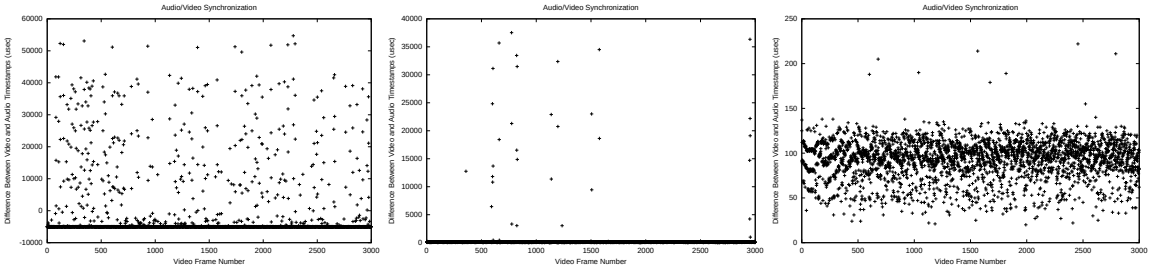
Non-kernel CPU load Figure 2 shows the audio/video skew in mplayer on Linux and on TSL when a CPU stress test is the competing load. This competing load runs an infinite loop consuming as much CPU as possible. Figure 2(a) shows that for standard Linux the maximum skew is large and ranges from -5 ms to 50 ms when the X server is run at a non-real time priority. Although not shown here, mplayer compensates for Linux's 10 ms timer resolution, and hence the skew under Linux without any competing load lies between -5 ms to 5 ms. Figure 2(b) shows that the skew for TSL, when X is run at a non-real time priority, is still as large as 35 ms. However, in this case, mplayer does not have to compensate for coarse timer resolution and thus most skew points lie close to 0 ms. Finally, Figure 2(c) shows that the skew for TSL improves considerably and is less than 250 us when the X server runs at real-time priority. The real-time priority value of X is the same as the priority assigned to mplayer.

These figures show that TSL 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. Note that Linux with the X server at real-time priority still has a skew between -5 ms to 5 ms because of the timer resolution (not shown here).

As a result of this experiment, the rest of the experiments in this section are run with both mplayer and X 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 TSL when the background load copies a large 8 MB memory buffer to a file with a single `write` system call. Figure 3(a) shows that the audio/video skew is as large as 90000 μs 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 μs for TSL. This improvement occurs as a result of improved kernel preemptibility for large write calls in TSL.

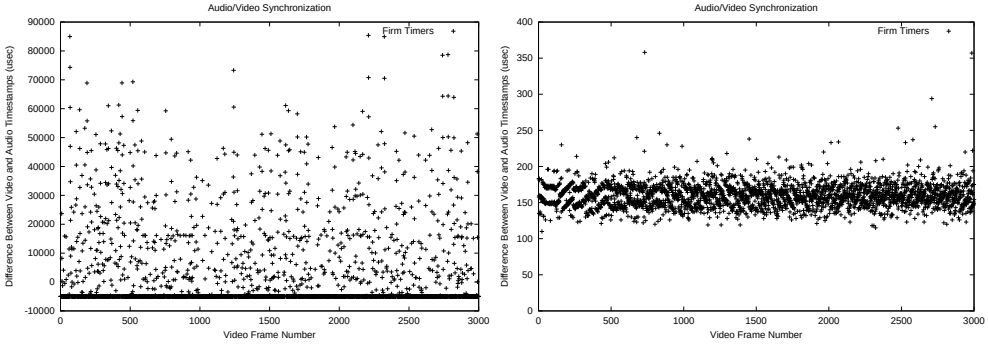
File System Load The third experiment compares the audio/video skew in mplayer between Linux and TSL when the background load repeatedly copies a compiled Linux kernel sources directory recursively and then flushes the file system. This directory has 13000 files and 180 MB of data and is stored on the 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 12000 μs while Figure 4(b) shows that skew is less than 500 us under TSL. This result shows that TSL can provide low latencies even under heavy file-system and disk load.



a) Linux, X server non-real time (b) TSL, X server non-real time (c) TSL, X server real-time

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.

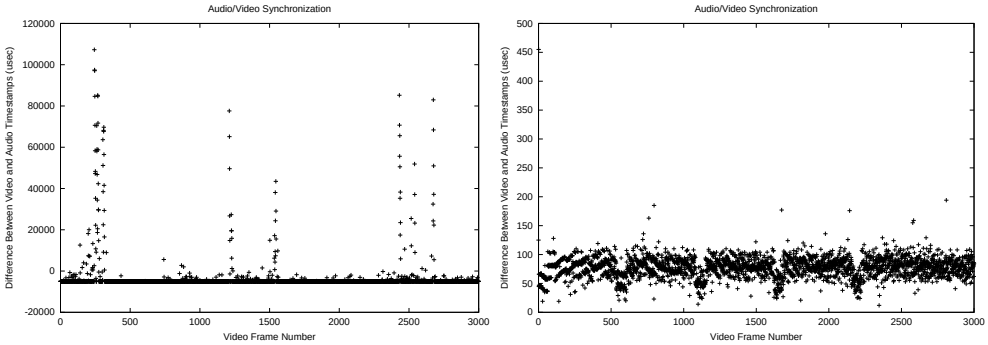
Figure 2: Audio/Video Skew on Linux and on TSL with user-level CPU load.



(a) Linux (b) TSL

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).

Figure 3: Audio/Video Skew on Linux and on TSL with kernel CPU load.



(a) Linux (b) TSL

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).

Figure 4: Audio/Video Skew on Linux and TSL with file-system load.

Comparison with Linux-SRT We also performed the previous experiments on Linux-SRT [6], which improves support for real-time applications by providing finer-grained timing than standard Linux, a reservation scheduler, and a modification to the X server to prioritize graphics rendering based on the scheduling parameters of tasks. Figure 5 shows the results of the experiments. For the non-kernel CPU load, the audio/video skew in Linux-SRT, when the X server was run at real-time priority, was generally less than 2 ms although the worst case latency is 7 ms. In this test, latency is dominated by timer latency, which is 1 ms on Linux-SRT. With the kernel CPU load test, the worst case latency was 60 ms, while the file-system test produced worst case latencies of 30 ms. These latter tests stress kernel preemption while Linux-SRT is a non-preemptive kernel. These results show that real-time scheduling and more precise timers are insufficient for time-sensitive applications, and that a responsive kernel is also required.

4.2.2 Proportion-Period Scheduler

As explained earlier, the proportion-period scheduler provides temporal protection when multiple time-sensitive tasks are run together. Our original motivation for implementing the TSL system was to provide an accurate implementation of a proportion-period scheduler. We use this scheduler to provide a fine-grained reservation mechanism for a higher-level feedback-based real-rate scheduler [20]. The problem with standard proportion-period scheduling is that it is difficult to correctly estimate a thread's proportion and period requirements in a general-purpose environment. To address this problem, 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 accuracy of allocating resources using a feedback controller 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 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.

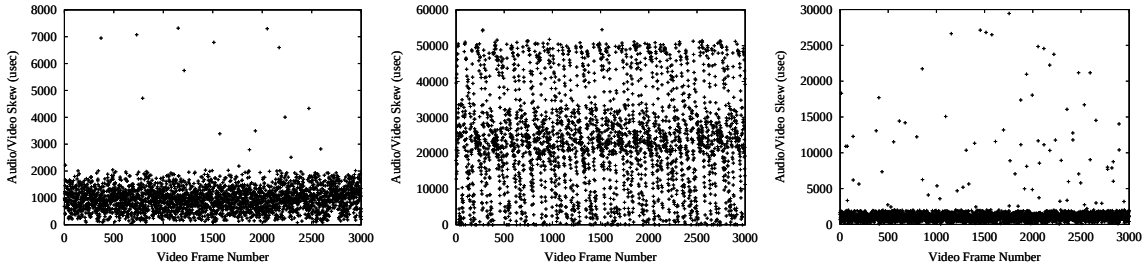
The proportion-period scheduler implementation on TSL uses firm-timers for implementing period boundaries and proportion policing. To evaluate the accuracy of this scheduler when multiple time-sensitive applications are scheduled together, we ran two time-sensitive processes with proportions of 40% and 20% and periods of 8192 μs and 512 μs respectively.[2] 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.

Table 1 shows the maximum (not the average) deviation in the proportion allocated and the period boundary for each of the two processes over the entire experiment. This table shows that the proportion-period scheduler allocates resources with a very low deviation of less than 25 μs on a lightly loaded system. Under heavy 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. The maximum period deviation of 534 μs gives a lower bound on the latency tolerance of time-sensitive applications. For example, soft modems require periodic processing every 4 ms to 16 ms [8] and thus could be supported on TSL at the application level even under heavy file system load.

Figure 1 shows that while the maximum amount of time stolen is 490 μs and 20 μs for tasks 1 and 2, this time is over different periods. In particular, the interrupt handling code steals a maximum of 4-6% allocation time from the proportion-period processes. Note that this stolen time is due to interrupts other than the APIC timer interrupts since we maintain precise CPU cycle counter times for accounting purposes.

Currently, we provide the value of the stolen time to proportion-period applications or our feedback scheduler so that they can deal with it, for example by increasing thread allocations. This choice was motivated by

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



(a) User-level CPU load (b) Kernel CPU load (c) File-system load

Figure 5: Audio/Video Skew on Linux-SRT with user-level CPU load, kernel CPU load and file-system load.

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

Table 1: Deviation in proportion and period when two processes are run under the proportion-period scheduler in TSL.

the idea that the feedback scheduler should be informed about scheduling "errors" as much as possible, which was especially important because, before TSL, these errors were significantly larger. In the future, we plan to investigate improving the performance of proportion-period scheduling in the presence of heavy file system load by explicitly scheduling interrupt processing.

4.3 System Overhead

In this section, we focus on the performance overheads of TSL. There are two main sources of overhead in TSL as compared to standard Linux: 1) the cost of executing code at the newly inserted preemption points, and 2) the cost of executing firm timers.

4.3.1 Checking for Preemption

At each preemption point, there is a cost associated with checking for preemption, and then if scheduling is needed, there is a cost for executing preemption. We do not explicitly measure the second cost because it depends on the workload. However, one instance where we expect that more preemption will occur in TSL is when firm timers are used, since firm timers can cause preemption at a finer granularity. Hence, we discuss this cost later when we present the overhead of firm timers.

We measured the cost of the additional preemption checks in TSL by running a set of benchmarks that are known to stress preemption latency in Linux [1]. In particular, we ran three separate tests, a memory access test, a fork test and a file-system access test. Note that these tests are designed to stress preemption checks and thus measure their worst-case overhead. We expect that these checks will have a smaller impact on real applications. The memory test sequentially accesses a large integer array of 128 MB and thus produces several page faults in succession. The fork test creates 512 processes as quickly as possible. The file-system test repeatedly copies data from a 2 MB user buffer to a file that is 8 MB long and flushes the buffer cache. By running these tests, we expect to hit the various additional preemption checks that exist in TSL as compared to Linux. We measured the ratio of the completion times of these tests under TSL and under Linux in single user mode. Since no other process is running, these tests do not cause additional preemption and thus we are able to evaluate the cost of checking the additional preemption points. Firm

timers were disabled in this experiment because we did not want to measure the cost of checking for soft timers.

The memory test under TSL has an overhead of 0.42 ± 0.18 percent while the fork test has an overhead of 0.53 ± 0.06 percent. The file system test did not have a significant overhead (in terms of confidence intervals). These tests indicate that the overhead of checking for preemption points in TSL compared to standard Linux is very low.

4.3.2 Firm Timers

The second cost in TSL is associated with executing firm timers. Firm timers provide an accurate timing mechanism which allows high frequency timer programming. However, increasing the timer frequency can increase system overhead because each timer event can cause a one-shot timer interrupt, which results in cache pollution. To mitigate this overhead, our firm timers implementation combines one-shot (or hard) timers and soft timers. In this section, we present experiments to highlight the advantages of firm timers as compared to hard timers and show that the overhead of firm timers on throughput-based applications is small even when firm timers are used heavily.

The cost of firm timers can be broken into three parts: 1) costs associated with hard timers exclusively, 2) costs that hard and soft timers have in common, and 3) costs associated with soft timers exclusively. The first cost occurs due to interrupt handling and the resulting cache pollution. The second cost lies in manipulating and dispatching timers from the timer queue and executing preemption for an expired timer thread. The third cost is in checking for soft timers. Note that the cost of executing preemption is present in both cases and thus the experiments presented below account for this cost when firm timers are used. Based on this breakup, it should be obvious that the soft timing component of firm timers will have lower overhead than hard timers if the cost for checking for timer expiry is less than the additional cost of interrupt handling in the pure hard timer case. This relation is derived in more detail below.

We will first compare the performance overhead of firm timers under TSL versus standard timers in Linux. This comparison is performed using multiple applications that each require 10 ms periodic timing. This case is favorable to Linux because the periodic timing mechanism in Linux synchronizes all the timers and generates one timer interrupt for all the threads, although at the expense of timer latency. In contrast, firm timers provide accurate timing but can generate multiple interrupts within each 10 ms period. Then we will evaluate the performance of firm timers for applications that require tighter timing than standard Linux can support.

In the following experiments, we measure the execution time of a throughput-oriented application when one or more time-sensitive processes are run in the background to stress the firm timers mechanism. We implement a time-sensitive process as a simple periodic task that wakes up on a signal generated by a firm timer (using the `setitimer()` system call), measures the current time and then immediately goes to sleep each period. In the rest of this section, we refer to this task as a timer process. For the throughput application, we selected `povray`, a ray-tracing application and used it to render a standard benchmark image called `skyvase`. We chose `povray` because it is a compute intensive job with a large memory footprint. Thus our experiments account for the effects of cache pollution due to the fine-grained timer interrupts. The performance overhead of firm timers is defined as the ratio of the time needed by `povray` to render the image in TSL versus the time needed to render the same image in standard Linux.

Comparison with Standard Linux We first compare the performance overhead of firm timers on TSL with standard timers running on Linux. To do so, we run timer processes with a 10 ms period because this period is supported by the tick interrupt in Linux. As explained above, we expect additional overhead in the firm timers case because, unlike with the periodic timers in Linux, the expiration times of the firm timers are not aligned. To stress the firm timers mechanism and clearly establish the performance difference, we ran two experiments with a large number of 20 and 50 timers processes.

Figure 6 shows the performance overhead of firm timers as compared to standard Linux timers when 20 timer processes are running simultaneously. This figure shows the overhead of TSL with hard timers, firm timers with different overshoot values (0 μs, 50 μs, 100 μs, 500 μs) and pure soft timers. Each experiment was run 8 times and the 95% confidence intervals are shown at the top of each bar. The figure shows that pure soft timers have an insignificant overhead compared to standard Linux while hard and firm timers have a 1.5 percent overhead. In this case, increasing the overshoot parameter of firm timers produces only a small improvement.

Figure 7 shows the results of the same experiment but with 50 timers. Once again soft timers have an insignificant overhead. In addition, the decrease in overhead of firm timers with increasing overshoot is more pro-

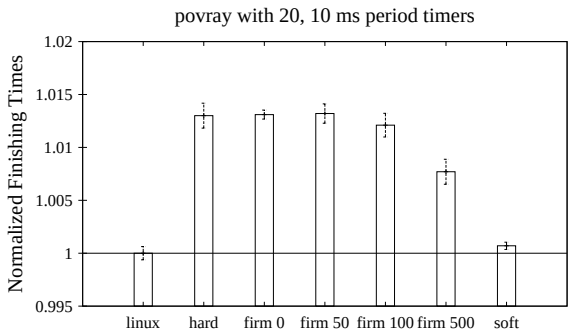


Figure 6: Overhead of firm timers in TSL as compared to standard Linux with 20 timer processes.

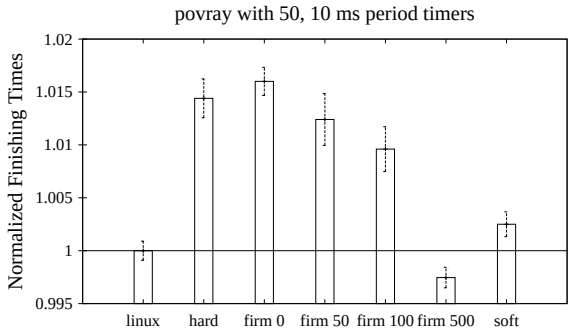


Figure 7: Overhead of firm timers in TSL as compared to standard Linux with 50 timer processes.

nounced in this case. The reason is that with increasing number of timers, timers are more likely to fire due to soft timers than the more expensive hardware APIC timer.

Interestingly, in Figure 7, the `povray` program completes faster on TSL with 500 μs firm-timer overshoot than on a standard Linux kernel. The reason for this apparent discrepancy is that the standard Linux scheduler does not scale well with large numbers of processes. On Linux, all 50 processes are woken at the same time (at the periodic timer interrupt boundary) and thus Linux has to schedule 50 processes at the same time. In comparison, on a firm timers kernel the 50 timers have precise 10 ms expiration times and are not synchronized. Hence, the scheduler generally has to schedule one or a small number of processes when a firm timer expires. In this case, the overhead of the scheduler on standard Linux dominates the overhead of the firm timers mechanism in TSL.

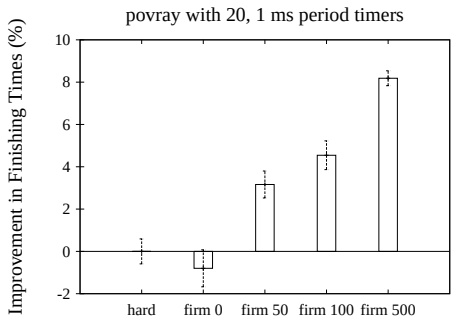


Figure 8: Comparison between hard and firm timers with different overshoot values on TSL.

Overhead at High Frequencies We also performed the same experiment but with periodic processes running at higher frequencies to simulate time-sensitive applications that have periodic timing requirements tighter than standard Linux can support. Figure 8 shows the improvement in time 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. Similarly, pure soft timers are not shown in this figure because they do not guarantee that each timer fires every 1 ms. This figure shows the improvement in finishing time of `povray` with firm timers with different overshoot values compared to hard timers. The benefit of the firm timers mechanism for improving throughput becomes more obvious with increasing overshoot when the process periods are made shorter. For example, there is an 8% improvement with a 500 μs overshoot value while the corresponding improvement in Figures 6 and 7 is 0.5% and 1.6%.

Discussion 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.

The previous behavior can be explained as follows. Let N_t be the total number of timers that must fire in a given interval of time, N_h the number of hard timers that fire, N_s the number of soft timers that fire (hence, $N_t = N_h + N_s$) and N_c 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. Note that we described the components of these costs in the beginning of this section. The total cost of firing firm timers is $C_c N_c + C_h N_h + C_s N_s$. If pure hard timers are used then the cost is $C_h N_t$. Hence, firm timers reduce overhead if $C_c N_c + C_h N_h + C_s N_s < C_h N_t$. After substituting $N_t = N_h + N_s$, this equation simplifies to $N_s/N_c > C_c/(C_h - C_s)$.

Hence, 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 overhead of one-shot timers. From our experiments, we have extrapolated that $C_h = 8$ μs, $C_s = 1$ μs, and $C_c = 0.15$ μs, hence the firm timers mechanism becomes effective when $N_s/N_c > 0.15/(8-1) = 0.021$, or when more than 2.1% of the soft timer checks result in the firing of a soft timer.

Note that the number of checks N_c depends on the amount of interrupts and system calls that occur in the machine, whereas the number of soft timers that fire N_s depends on how the checks and the timers' deadlines are distributed in time and the overshoot value. Aron and Druschel's original work on soft timers [2] 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 N_s/N_c), although how close is very workload dependent. Firm timers have the benefit of assuring low timer latency even for workloads with poor distributions, yet retaining the performance benefits of soft timers when the workload permits.

Note that soft timer checks are normally placed at kernel exit points where kernel critical sections end and where the scheduler function can be invoked. The use of a preemptible kernel design in TSL reduces the granularity of non-preemptible sections in the kernel and potentially allows more frequent soft timer checks at the end of spinlocks and hence can provide better timing accuracy. The key issue here is the overhead of this approach, which depends on the ratio N_s/N_c, i.e., whether sufficient additional soft timers fire as a result of the additional soft checks. While our current firm timer implementation does not check for timers at the end of each spinlock, we plan to evaluate this approach in the future.

5 Conclusions

This paper describes the design and implementation of a Time-Sensitive Linux (TSL) 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 TSL are firm timers for accurate timing, fine-grained 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 TSL on throughput-oriented applications is low and thus such a system is truly a general-purpose system since it can be used effectively for both time-sensitive and traditional interactive and long-running batch applications.

Although the first results presented in this paper are promising, TSL still need further investigation, since there are open issues related to interrupt service times, fine-grained accounting of time, latencies caused by network processing, and firm timers performance. For firm timers in particular, we are interested in investigating whether real workloads commonly lead to the $N_s/N_c > K$ condition under which firm timers are effective.

Acknowledgments

Andrew Black, Wu-chi Feng and Wu-chang Feng provided many useful suggestions on the initial draft of the paper. We would like to thank the reviewers, especially our shepherd, Timothy Roscoe, for taking the time and suggesting numerous improvements to the paper.

References

[1] 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 *Real Time Technology and Applications Symposium (RTAS)*, September 2002.

[2] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, August 2000.

[3] T. P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, pages 67–99, 1991.

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

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

[6] Stephan Childs and David Ingram. The Linux-SRT integrated multimedia system: Bringing QoS to the desktop. In *Real-Time Technology and Applications Symposium*, May 2001.

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

[8] Michael B. Jones and Stefan Saroiu. Predictability requirements of a soft modem. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Cambridge, MA, June 2001.

[9] 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.

[10] 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.

[11] Robert Love. The Linux kernel preemption project. `http://kpreempt.sf.net`.

[12] Montavista Software - Powering the embedded revolution. `http://www.mvista.com`.

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

[14] Mplayer - Movie player for linux. `http://www.mplayerhq.hu`.

[15] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *NOSSDAV*, November 1993.

[16] Jason Nieh and Monica Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Symposium on Operating Systems Principles*, October 1997.

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

[18] S. Savage and H. Tokuda. RT-Mach timers: Exporting time to the user. In *USENIX 3rd Mach Symposium*, April 1993.

[19] 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.

[20] David Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation*, February 1999.

[21] The X window system. `http://www.x.org`.

[22] Yu-Chung and Kwei-Jay Lin. Enhancing the real-time capability of the Linux kernel. In *IEEE Real Time Computing Systems and Applications*, October 1998.