Operating System Support for Low-Latency Streaming

Ashvin Goel
B.S., I.I.T Kanpur, India, 1992
M.S., UCLA, 1996

A dissertation presented to the faculty of the
OGI School of Science & Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

July 2003

The dissertation "Operating System Support for Low-Latency Streaming" by Ashvin Goel has been examined and approved by the following Examination Committee:

Jonathan Walpole
Professor, OGI School of Science and Engineering
Thesis Research Adviser

Calton Pu
Professor, Georgia Institute of Technology

Mark Jones
Associate Professor, OGI School of
Science and Engineering

Wu-chang Feng
Assistant Professor, OGI School of
Science and Engineering

Dedication

To my parents, whose wholehearted support helped me start this endeavor.

To Nadia, my love, whose wholehearted support helped me complete it.

Acknowledgments

The ideas in this dissertation, its character and its form, have been heavily influenced by a long list of people with whom I have been fortunate enough to collaborate during my PhD. Jon Walpole, my advisor, has been instrumental in the successful completion of this work. He provided an environment that nurtured my growth during the early stages of the PhD process, while giving sufficient freedom and responsibility during the later stages of my PhD. His invaluable advice and his philosophy about keeping a balance between work, hobbies and personal life inspired me and will continue to inspire me to do my best.

My confidence as a researcher grew under my previous advisor, Calton Pu, who fully supported and encouraged me at each stage of my work. Later, Molly Shor and David Steere motivated my interest in using control theory for software systems. I have enjoyed interacting with Charles Consel who sparked my interest in using programming language techniques for systems work. OGI would not be the same without the Feng brothers, W2 and W4, whose arrival has energized our SYSL group and made the work environment a lot more fun and productive. I wish they had come to OGI earlier!

During my Ph.D, I collaborated extensively with my office mate Buck Krasic. Our daily discussion about every aspect of computer science helped me shape and strengthen my views about what is important in software systems. Luca Abeni, a visiting Ph.D. student from Italy, showed me the power of story writing in scientific papers! I could always count on Kang Li, our resident networking expert. I would like to thank many other students and interns who made OGI a fun environment, including Jie Huang, Dan Revel, Anne-Francoise Le Meur, Jim Snow, Francis Chang, Mike Shea and Brian Code.

Contents

List of Tables

List of Figures

Abstract

Operating System Support for Low-Latency Streaming

Ashvin Goel

Supervising Professor: Jonathan Walpole

Streaming applications such as voice-over-IP and soft modems running on commodity operating systems (OSs) are becoming common today. These applications are characterized by tight timing constraints that must be satisfied for correct operation. For example, voice-over-IP systems have one-way delay requirements of 150-200 ms where only a third of that time can be spent in the operating system and in the application layers. Similarly, soft modems require periodic execution with low jitter every 12.5 ms from the operating system. To satisfy the timing requirements of these low-latency streaming applications, the operating system (OS) must allow fine-grained scheduling so that applications can be scheduled at precisely the time they require execution. Unfortunately, the metric traditionally used to evaluate current general-purpose OSs focuses on application throughput rather than latency. Consequently, current operating systems use mechanisms such as coarse-grained timers and schedulers, and large packets to amortize operating system "overhead" over long periods of application-level work. This approach has the effect of improving throughput at the expense of latency. Similarly, current OSs have large non-preemptible sections and use large kernel buffers to improve CPU throughput. All of these mechanisms result in increased latency in the kernel, which conflicts with the timing requirements of low-latency applications because it reduces control over the precise times at which applications can be scheduled.

This dissertation shows that general-purpose OSs can be designed to support low-latency applications without significantly affecting the performance of traditional throughput-oriented applications. We identify and experimentally evaluate the major sources of latency in current OSs and show that these latencies have three main causes: *timing mechanism*s, *non-preemptible kernel sections* and *output buffering*. We propose three techniques, *firm timers*, *fine-grained kernel preemptibility*, and *adaptive send-buffer tuning* for reducing each of these sources of latency. The firm timers mechanism provides accurate timing with low overhead. We use fine-grained kernel preemptibility to obtain a responsive kernel. Adaptive send-buffer tuning helps to minimize buffering delay in the kernel for TCP-based streaming applications. These techniques have been integrated in our extended version of the Linux kernel, which we call Time-Sensitive Linux (TSL). Our evaluation shows that TSL can be used by streaming applications with much tighter timing constraints than standard Linux can support, and that it does not significantly degrade the performance of throughput-oriented applications.

Low kernel latency in TSL enables fine-grained, feedback-based scheduling for supporting the needs of low-latency applications. Traditionally, real-time scheduling mechanisms have been used to provide predictable scheduling latency. However, these mechanisms are difficult to use in general-purpose OSs because they require precise specification of application requirements in terms of low-level resources. Such a specification may not be statically available in this environment. Hence, this thesis presents the design, implementation and evaluation of a novel feedback-based real-rate scheduler that automatically infers application requirements and thus makes it easier to use real-time scheduling mechanisms on general-purpose OSs. To be effective, feedback scheduling requires fine-grained resource monitoring and actuation. TSL, unlike conventional OSs, supports both these requirements because it provides precise timing control. Such control has enabled us to implement time-sensitive applications such as `gscope`, a software oscilloscope, and `TCP`ivo, a software network traffic generator.

Chapter 1

Introduction

In the last decade, CPU speeds have increased tremendously leading to an abundance of processing power on desktop computers. Consequently, today it is hard to saturate commodity processors with a common mix of desktop applications like word processing, web browsing, email, on-line radio, video streaming, etc. These processors have become fast enough to process applications that were normally reserved for dedicated hardware. For example, desktop CPUs can be used to run a soft modem application, where the main processor executes modem functions that have traditionally been performed in modem hardware. Similarly, multimedia applications such as video streaming and conferencing do not require dedicated hardware anymore. Another application enabled by today's fast processors is a software network-traffic generator. A typical traffic generator such as the IXIA [52] is implemented in hardware and can generate and replay packet traffic with precise timing. Today, it is possible to implement such a generator in software. The characteristic feature of these applications is that they are driven by real-world events and have tight timing constraints that must be satisfied for correct operation. For example, with live video streaming, frames need to be captured periodically (every 33.3 ms for a 30 frames per second video), transmitted across the network and displayed every 33.3 ms with a jitter of less than 2-3 ms or else video quality is impaired.

Although modern commodity hardware is ready, current general-purpose operating systems (OSs) don't provide good support for applications with tight timing guarantees because they focus on traditional throughput-oriented applications. To improve application throughput, OSs manage resources at a coarse granularity and amortize system overhead over long periods of application-level work. For example, they use a coarse scheduling quantum to reduce context-switch overhead and transfer data in large packets or disk blocks to reduce interrupt overhead. This approach

improves throughput but at the expense of timing guarantees to applications. We believe that the traditional goal of optimizing system and application throughput by squeezing every cycle at the expense of all others metrics, such as timing guarantees, has diminishing returns given the abundant processing power available today.

In this thesis, we address the need to integrate fine-grained timing behavior in general purpose OSs with the goal of supporting time-sensitive applications on commodity processors. This approach has several benefits including cost and flexibility. For example, a hardware-based network-traffic generator can be replaced by a software traffic generator which helps reduce costs. The flexibility advantages come from a software versus a hardware implementation. For example, enhancements such as additional diagnostics capability or bug fixes in the traffic generator implementation can be performed more easily in software compared to in hardware.

The choice of a commodity general-purpose OS instead of a dedicated system for time-sensitive applications has several additional benefits. Commodity OSs are available inexpensively and thus supporting these applications on them is a cheaper alternative than building a new OS or end-host hardware such as a video conferencing box that is exclusively designed for such applications. In terms of software engineering, commodity OSs are considered easier to maintain than dedicated systems due to their large user and developer base. In addition, the writing and maintaining software for a well-known commodity OS API has a shorter learning curve. In particular, this approach allows integrating time-sensitive applications with traditional applications since both types of applications use the same API.

This thesis aims to support time-sensitive as well as throughput-oriented applications on a general-purpose OS and provide good performance to both classes of applications. Today, users are accustomed to traditional applications, such as FTP and web access, on their desktop. The novelty of our approach is to enable time-sensitive applications, such as video conferencing, video surveillance and network-traffic generators, on the general-purpose desktop computer.

Another goal of this thesis is to explore easy-to-use programming models for time-sensitive applications. Current general-purpose OSs provide a simple virtual machine API to applications where each application executes on the virtual machine assuming no other applications are present. Although this API is easy to use, it provides applications with little control over when and how often they will be executing.

For time-sensitive applications, an API that allows these applications to express their temporal constraints to the OS is needed. The real-time community has studied this problem extensively [67, 75, 101] and thus real-time systems provide such APIs and the OS provides fine-grained execution control. We borrow some ideas from real-time systems to provide timing control in general-purpose OSs. The main problem with real-time APIs is that they require precise and low-level specification of an application's timing requirements in terms of resources such as CPU capacity. In a general purpose environment, obtaining such a specification is non-trivial because an application's resource needs depend on the mix of other applications running on the system and, in addition, can be data and processor dependent. In this thesis, we present the design, implementation and evaluation of a novel adaptive scheduler that uses *feedback-control* to automatically infer application requirements and thus makes it easier to use real-time scheduling mechanisms on general-purpose OSs for time-sensitive applications. To be effective, this feedback approach requires fine-grained resource monitoring and actuation. Interestingly, these requirements make the feedback scheduler itself a time-sensitive application.[1] Hence, the requirements it imposes on the OS are similar to the requirements of other time-sensitive applications. We discuss this issue further in Section 1.2.2.4.

In the next section, we describe the requirements that time-sensitive applications impose on OSs and explain how current OSs do not adequately satisfy these requirements. Then, Section 1.2 describes our approach for meeting these requirements on general-purpose OSs. The contributions of this dissertation are presented in Section 1.3.

1.1 Low-Latency Application Requirements

Time-sensitive applications are driven by real-world input events such as wall-clock time or packet arrivals. Each event is intercepted by the kernel and delivered to the application. The application processes the event and generates an output response which is sent to the kernel. The kernel delivers this response to the external world. Figure 1.1 shows the sequence of these events. We define the latency from the arrival of an input event to when the application receives this event as *input latency*. Similarly, we define the latency from when the application finishes processing an event

[1] Here we refer to the scheduler as an application even though it may be implemented in the OS.

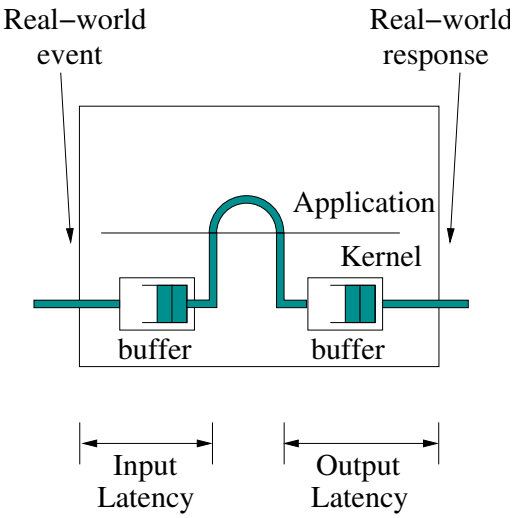


Figure 1.1: Input and output latency.

to its delivery to the external world as *output latency*. Both input and output latency are delays incurred in the kernel during the processing of an event. From now we refer to time-sensitive applications as *low-latency* applications because they require *low* input and output latency from the OS, which reduces the end-to-end latency from the arrival of an event to the delivery of its response. The following paragraphs explain these requirements with three illustrative examples of low-latency applications, a network traffic generator, a soft modem and a video conferencing application.

1.1.1 Network Traffic Generator

A network traffic generator helps to emulate network load and can be used to evaluate the design of network devices such as routers, switches and firewalls. For example, one could implement a network device using a network processor such as the IXP network processor [57] and evaluate it using a packet generator. One method for implementing a packet generator is to use a trace-driven approach, where a trace is collected and stored to disk using a tool such as `tcpdump` and then later replayed against the target device. When driven by a representative library of traces, such an approach is fast, reproducible and highly accurate in generating the modulation of packets as they pass through numerous hops in a network.

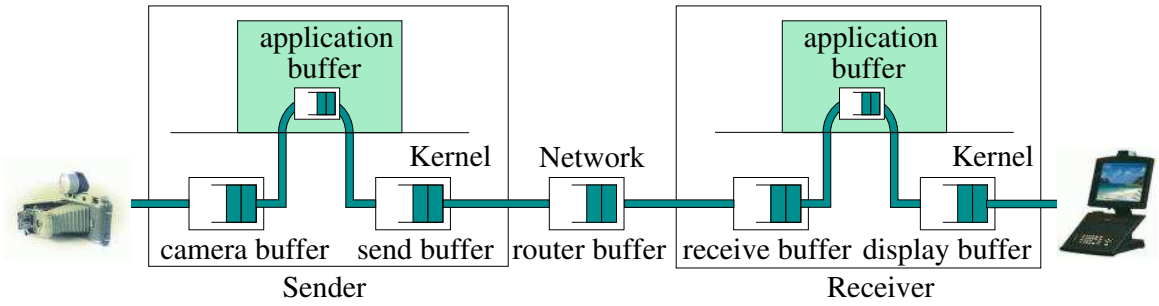
A trace-driven packet generator needs a high-performance packet replay engine (in addition to a packet collection engine). The replay engine must be able to replay packet traces with high performance and with high accuracy. For example, packet send events must be triggered with precise timing, which requires an accurate, low-overhead timing mechanism. In addition, the replay engine must be scheduled immediately when packets need to be sent. Finally, packets need to be sent out to the network with low output latency so that the original packet timing can be preserved.

1.1.2 Soft Modems

Soft modems use the main processor to execute modem functions that are traditionally performed by hardware on the modem card. Their cycle time for processing needs (i.e., the inter-arrival time between input events) must lie between 3 to 16 ms to avoid impairing modem audio and other multimedia audio applications [23]. Consequently, soft modems require low input latency so that they can be scheduled periodically with low jitter after the arrival of timer events. In addition, they need a minimum amount of CPU capacity within each cycle time. For example, Jones reports that soft modems require at least 14.7% CPU allocation on a 600 MHz Intel x86 processor [60].

Unfortunately, current commodity OSs have large input and output latency. For example, Linux, by default, provides 10 ms granularity timing. Hence, a soft modem can experience as much as 10 ms input latency after a wall-clock expiration event. This timing resolution is insufficient for a soft modem application, which needs to be run periodically with a precise period. Hence, a more precise timing mechanism is needed.

Jones [60] reports that soft modem hardware vendors implement modem processing in high-priority kernel interrupt service routines (ISRs) because an application-level implementation is not guaranteed the correct CPU capacity every cycle time and thus occasionally gets starved on a standard Windows 2000 OS. This starvation occurs when other resource hungry applications such as Internet Explorer are starting or when certain device drivers such as the IDE driver copy data in long non-preemptible sections. Unfortunately, the ISR implementation can starve other low-latency applications such as audio processing since ISRs run in non-preemptible sections. An application-level modem implementation alleviates this problem but requires a kernel that has an accurate timing mechanism, provides fine-grained preemptibility, and has better support for



End-to-end delay in this application occurs due to several factors including buffers along the pipeline path and processing times at each component.

Figure 1.2: An example of a low-latency video streaming application.

real-time scheduling algorithms such as a proportion-period scheduler.

1.1.3 Video Conferencing

Figure 1.2 shows the architecture of a video conferencing application. In this application, data arrives from the camera to the sender application, which then processes the data and then forwards it to the receiver. The receiver application does further processing and forwards the data to the display. Conferencing is a symmetric two-way process and thus the reverse path has the same architecture.

The end-to-end or camera-to-display delay requirements for this application are tight. The International Telecommunications Union G.114 document recommends 200 ms as the upper limit for one-way end-to-end delay for most interactive applications [51] of which about 100 ms is allocated for propagation delay. This delay is unavoidable when transmitting data over long distances such as from the West coast to the East coast of the US. Hence, the application and the OS have a budget of less than 100 ms for operations such as packet capture, encoding/decoding and jitter buffering. Of this delay, we assume that 60 ms are spent at the application level, 30 ms at the

sender and 30 ms at the receiver for encoding and decoding video data.[2] Then the total input and output latency in the OSs at the sender and the receiver must be less than 40 ms to avoid impairing the performance of interactive conferencing.

Video conferencing needs low input latency so that the application can either respond quickly after the arrival of camera data (on the sender side) or of packet data (on the receiver side). It requires low output latency so that data written by the application is either quickly transmitted to the network (on the sender side) or displayed in time (on the receiver side).

Unfortunately, current OSs, have high input latency. For example, our evaluation of standard Linux shows that it can have non-preemptible sections that are as long as 20-100 ms under heavy system load [2]. Under these conditions, low-latency applications don't get execution control in time, and the OS is unable to satisfy the 40 ms video conferencing delay requirement in the OSs at both the sender and the receiver sides. Later in this thesis, we show that the kernel needs to be more responsive to satisfy this requirement.

In addition to high input latency, current OSs can have high output latency. To demonstrate this behavior, we evaluated output latency on the sender side OS when streaming over TCP, the most common transport protocol on the Internet. Under heavy network load, output latency with TCP can be as large as 1-2 seconds even when the round-trip time is less than 100 ms [39]. The 200 ms end-to-end latency requirements of video conferencing cannot be met under these conditions. Our evaluation showed that most of the 1-2 second latency occurs as a result of send buffering in sender kernel. This buffering should be reduced to meet the latency requirements of TCP-based low-latency streaming applications such as video conferencing.

A third problem with commodity OSs for video conferencing is unpredictable CPU scheduling. For example, applications may not be able to encode, process or decode data in a timely manner since the OS does not make any scheduling guarantees. A real-time scheduler provides such guarantees but requires timing information in terms of low-level resources such as CPU allocation. This API is unsuitable in a general-purpose environment because the CPU requirements can change over time. For example, the CPU requirements for decoding a variable bit-rate video stream vary over time and thus cannot be easily specified statically. Given the timing requirements

[2]The inter-frame time in some common video standards is 33.3 ms. Our assumption of 30 ms for encoding and decoding (out of the 33.3 ms) is conservative because it would be less on a fast processor.

of low-latency applications, a method for automatically and dynamically deriving their resource requirements is needed.

1.2 Our Approach

This section describes our approach for supporting low-latency applications on general-purpose OSs. First, in section 1.2.1, we identify sources of input and output latency in an OS. We show that the timing mechanism, non-preemptible kernel sections and scheduling are the main sources of input latency. For applications that stream data using TCP, we show that the sender-side buffering in TCP is the main source of output latency.

In Section 1.2.2, we describe our basic solution, which consists of techniques that help reduce each of these sources of input and output latency. The design, implementation and evaluation of these techniques is presented in more detail in later chapters. We have integrated these techniques in the Linux OS, and we call the resulting system Time-Sensitive Linux (TSL). TSL is a general-purpose OS designed for implementing, running and evaluating the performance of low-latency applications. Several techniques in TSL are being incorporated in the standard Linux distribution and hence we expect that in the near future these techniques will become part of a commodity OS.

In Section 1.2.2.4, we motivate the need for a feedback-based scheduler that automatically infers the resource requirements of low-latency applications. Such a scheduler is itself a low-latency application and hence can be supported well on TSL. Finally, Section 1.2.2.5 describes a software oscilloscope that we have implemented that helps in visualizing and debugging the behavior of low-latency applications.

1.2.1 Latencies in Operating Systems

The execution time-line of a low-latency application can be viewed as a sequence of real-world input events that are delivered by the kernel to the application, which processes them to generate responses. Figure 1.3 shows one such event and the actions or steps that occur in the system as a result of this event until its response.

The real-world event can either be *time-driven* or *data-driven* as shown in the left of the figure. Time-driven events are triggered by wall-clock time. An example of a system that uses time-driven

events is a polling system such as a soft modem that periodically polls for data. Data-driven events are triggered by the arrival of data, such as video data from a video capture card or the arrival of network packets.

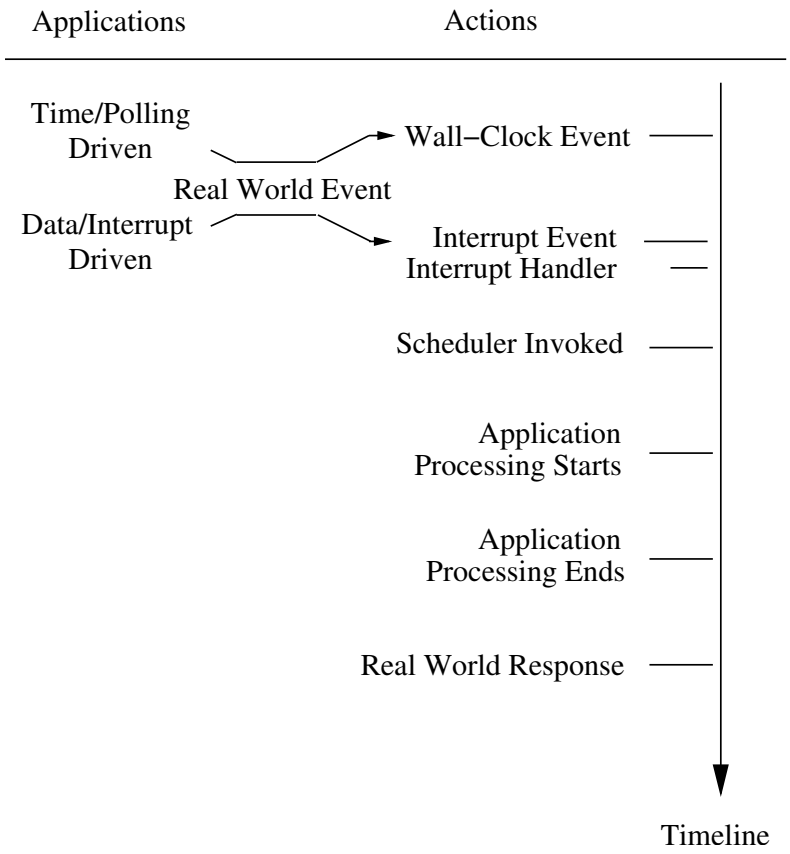


Figure 1.3: Execution time-line of a low-latency application.

The role of the OS is to "deliver" the timing or the data input events to the application and to "deliver" application generated output data to the external world. This delivery process consists of several steps within the OS. These steps are shown in the second column of Figure 1.3. Both time- or data-driven events eventually cause an interrupt. The OS handles this interrupt in the interrupt handler. Next, the OS invokes a scheduler to execute *some* application. Eventually, the scheduler chooses the low-latency application which then starts processing. When the application finishes processing, it sends data to the OS where the data is buffered until the real-world response such as data display or data transmission occurs.

Each of these OS steps cause latency as shown in the second column in Figure 1.4. As defined earlier, input latency is the time between the generation of the external event and the time when the low-latency application is scheduled. Output latency is the time between when the application generates output and the time when this output is delivered to the external world. Note that the OS or the application (or both) usually buffer data to hide the effects of input and output latency, and the higher these latencies, the larger the buffering needs. Note also that we use the terms input and output from the application's point of view.

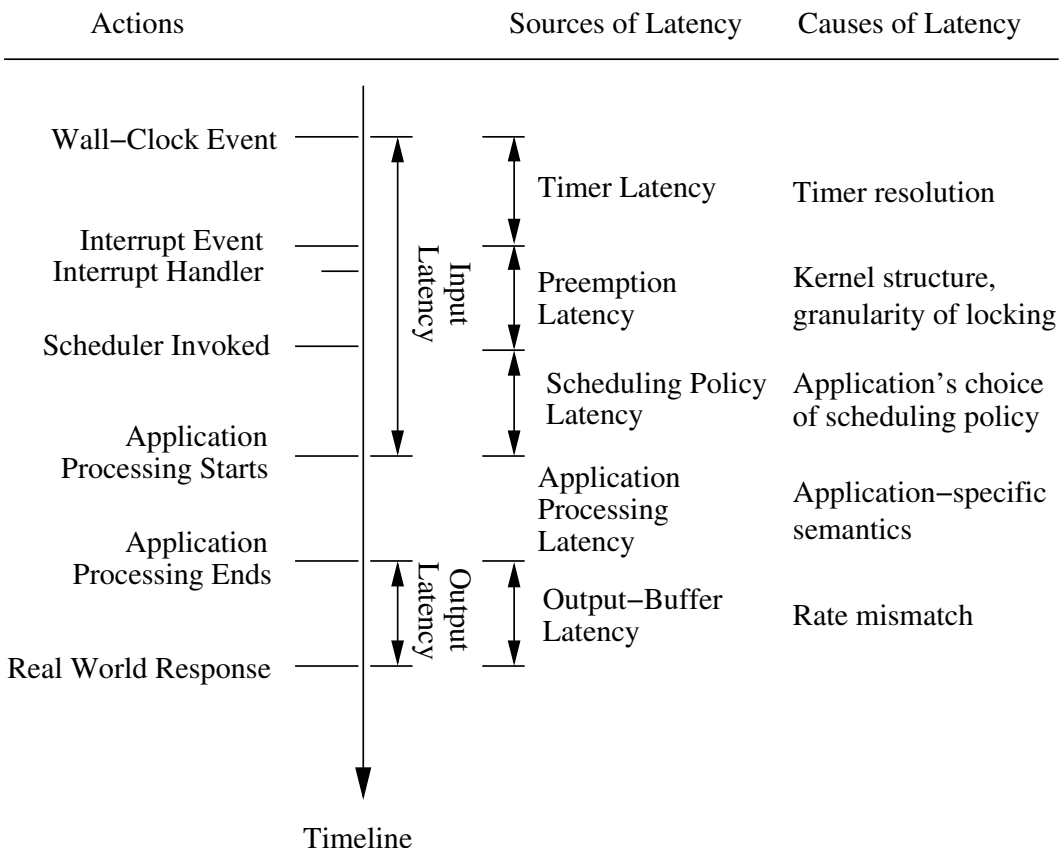


Figure 1.4: Execution time-line of a low-latency application.

As the figure shows, input latency is composed of three components: *timer latency*, *preemption latency* and *scheduling-policy latency* (or *scheduling latency*). Output latency can have various components such as preemption latency, file-system latency and network device latency. However, in this thesis, we will focus on TCP-based network streaming, and in this case, the most significant

component of output latency occurs due to a rate mismatch between the application's data rate and the rate at which network data transmission can occur. The output buffer accumulates data during this latency and hence we call the resulting latency, *output-buffer latency*.

The third column of Figure 1.4 shows the causes of these latencies. Below, we describe the components of input and output latency and their causes in more detail.

1.2.1.1 Timer Latency

Coarse-granularity timer resolution is the largest source of latency in commodity OSs such as Linux [2]. For example, Linux by default provides 10 ms granularity timing to kernel and user-level applications because kernel timers, which wake a sleeping application, are generally implemented using a periodic timer interrupt. Hence, a thread that sleeps for an arbitrary amount of time will experience as much as 10 ms latency if its expiration event is not on a timer-tick boundary. This timing resolution is insufficient for implementing low-latency applications such as video conferencing.

1.2.1.2 Preemption Latency

An accurate timing mechanism is necessary but not sufficient for reducing latency. For example, even if a timer interrupt is generated by the hardware at the correct time, an application may still run much later because the kernel is unable to interrupt its current activity, either because the interrupt is disabled or because the kernel is in a non-preemptible section. Our evaluation shows that preemption latency can be as large as 50-100 ms due to long execution paths in a general-purpose OS such as Linux [2]. The size of non-preemptible sections in general-purpose OSs has to be reduced to support low-latency applications.

1.2.1.3 Scheduling Latency

The scheduling policy used by a thread causes additional latency because a thread may not have the highest priority and thus is not scheduled immediately even if accurate timers and preemptible kernel features ensure that it enters the scheduler's ready queue at the correct time. The scheduling problem has been extensively studied by the real-time community. Real-time schedulers, such as a proportion-period scheduler, can provide low and predictable scheduling latency when used

appropriately but most such schedulers rely on strict assumptions such as the full preemptibility of threads for correctness. A kernel with short non-preemptible sections and 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.

1.2.1.4 Output-Buffer Latency

Output latency, in general, can be caused by some of the same factors as input latency. For example, long non-preemptible kernel sections can increase output latency. However, in this thesis, we only focus on output latency in network streaming applications that use TCP. With TCP, output-buffer latency can be very large, in the order of seconds, even when the network round-trip time is less than 100 ms [39]. Hence this latency masks the other components of end-to-end latency. Output-buffer latency occurs because TCP uses a large output buffer that can accumulate large amounts of data before the application is able to detect the problem and adapt its data rate. A larger buffer improves throughput but increases latency also. For low-latency applications, it should be possible to automatically tune the size of this buffer and hence trade throughput for lower latency.

1.2.2 Time-Sensitive Linux

This section introduces our solutions for supporting low-latency applications on general-purpose OSs. We propose four specific techniques, firm timers, fine-grained kernel preemptibility, adaptive send-buffer tuning and real-rate scheduling for reducing timer latency, preemption latency, output-buffer latency and scheduling latency respectively, as described in the previous section. We have integrated these techniques in the Linux kernel to implement Time-Sensitive Linux.

1.2.2.1 Firm Timers

Traditionally, general-purpose operating systems have implemented their timing mechanism with a coarse-grained periodic timer interrupt. This approach has low overhead but the maximum timer latency can be as large as the timer period. To reduce timer latency, firm timers use one-shot timers, which can fire at precise wall-clock times. Thus we expect that firm timers have a resolution close to hardware interrupt processing times.

There are two main issues with using one-shot timers. First, they have to be reprogrammed at each timer event. Second, they can cause an increase in the number of interrupts compared to a coarse-grained periodic timer interrupt approach. While timer reprogramming was expensive on traditional hardware, it has become inexpensive today. For example, one-shot timers in modern x86 machines can be reprogrammed in a few cycles. Hence one-shot timers are much more viable today.

The key overhead for the one-shot timing mechanism in firm timers lies in fielding interrupts, which cause a context switch and cache pollution. To avoid this overhead, firm timers use soft timers (originally proposed by Aron and Druschel [9]). Soft timers avoid interrupts by checking for expired timers at strategic points in the kernel such as at system call, interrupt and exception return paths. These checks are called soft timer checks. When system workloads cause frequent soft timer checks, we expect the combination of cheap one-shot timer reprogramming and soft timers to provide an accurate timing mechanism with low overhead. Chapter 2 presents the design, implementation and evaluation of firm timers.

1.2.2.2 Fine-Grained Kernel Preemptibility

Long preemption latencies in general-purpose OSs are caused by long non-preemptible sections. To support low-latency applications, the size of these non-preemptible sections must be reduced. This problem can be addressed using various approaches. One approach that reduces preemption latency is explicit insertion of preemption points at strategic points inside the kernel [87, 79] so that a thread in the kernel explicitly yields the CPU to the scheduler after it has executed for some period of time. In this way, the size of non-preemptible sections is reduced. Another approach, used in most real-time systems, is to use a preemptible kernel design [68, 78] which allows multiple threads to execute within the kernel at the same time but requires all kernel data to be explicitly protected using mutexes or spinlocks. The size of non-preemptible sections in this case is reduced to the time for which spinlocks are held. A third approach builds on the second one and explicitly inserts preemption points within spinlocks when spinlocks are held for a long time.

In this thesis, we evaluate and compare these approaches to determine their effectiveness at reducing preemption latency. We compare these approaches using micro-benchmarks as well as

real applications. We expect that the third approach which combines the benefits of the previous two approaches will yield the best results. Chapter 3 describes and evaluates these approaches in detail. It also evaluates the overhead of checking and performing preemption in these approaches.

We have incorporated a preemptive kernel patch for Linux from Robert Love [68] into TSL. Our experiments with real applications on TSL in Chapter 4.7 show that a fine-grained preemptive kernel complements firm timers to improve the performance of low-latency applications.

1.2.2.3 Adaptive Send-Buffer Tuning

For TCP-based streaming, output-buffer latency occurs because there is a rate mismatch in the application's sending rate and TCP's transmission rate. TCP uses a send buffer to hide this rate mismatch. This buffer also keeps packets that are currently being transmitted for retransmission since TCP provides lossless packet delivery. The packets that are buffered to match rates add output-buffer latency but the packets for retransmission do not add any latency because they have already been transmitted. Based on this insight, we expect that if the size of the send buffer is tuned so that it only buffers packets that have to be retransmitted, then TCP will have little or no output-buffer latency. We have implemented this adaptive buffer sizing technique for TCP in TSL. Chapter 4 describes our implementation and evaluates the effectiveness of this technique in reducing output-buffer latency for TCP flows.

1.2.2.4 Feedback CPU Scheduling

The choice of scheduling algorithm affects the scheduling latency experienced by threads. Traditionally, real-time scheduling algorithms such as priority-based and proportion-period scheduling have been used to provide predictable and low scheduling latency. The integration of a fine-grained preemptibility with the firm timers mechanism in TSL allows an accurate implementation of such schedulers. Chapter 5 provides an overview of these algorithms and evaluates the accuracy of our proportion-period scheduler implementation under TSL.

Although real-time schedulers provide predictable scheduling latency, it is hard to use them in a general-purpose operating system environment. In particular, with proportion-period scheduling, applications are assigned a proportion of the CPU over a period of time, where the correct proportion and period are analytically determined by humans. Unfortunately, it is difficult to correctly

estimate an application's proportion and period needs statically.

To solve this problem, we develop a feedback-based technique for dynamically estimating the proportion and period needs of an application based on observing the application's *progress*. An application specifies its progress needs to the scheduler by using application-specific *time-stamps*. These time-stamps indicate the progress rate desired by the application. For example, a video application that processes 30 frames per second can time-stamp each frame 33.3 ms apart. The key idea in our feedback-based scheduler, which we call the *real-rate* scheduler, is to use these time-stamps to allocate resources so that the rate of progress of time-stamps matches real-time. The real-rate scheduler uses feedback to assign proportions and periods to threads automatically as the resource requirements of threads change over time.

TSL provides a simple time-stamp based API that allows low-latency applications to specify their progress needs and hence allows these applications to express their timing constraints to the operating system. The novelty of our approach lies in using application-specific metrics to measure progress (as opposed to resource-specific metrics such as CPU cycles) and a feedback controller that maps this progress to specific proportion and period requirements. Chapter 5 presents the design, implementation and evaluation of our real-rate scheduler.

1.2.2.5 A Software Oscilloscope

Modern processors have made multimedia and other low-latency applications such as DVD players, DVD and CD burners, TV tuners and digital video editing and conferencing software common on desktop computers running commodity OSs. Unfortunately, implementing and test these low-latency applications is non-trivial because existing tools for visualizing and debugging alter the timing behavior. For instance, a standard debugger stops an application and thus affects its timing behavior. Thus, debugging and visualization tools specifically designed for low-latency applications are needed.

Unlike the ad hoc tools used for visualizing and testing low-latency software, there exists a time-tested visualization tool in the hardware community: the *oscilloscope*. The invention of the oscilloscope started a revolution that allowed engineers and users to "see" sound and other signals, experience data, and gain insights far beyond equations and tables [55]. Today, an oscilloscope,

together with a logic analyzer, is used for several purposes such as debugging, testing and experimenting with various types of hardware that often have tight timing requirements. We believe that a similar approach can be applied effectively for visualizing low-latency software systems. As a result, we have developed a user-level software visualization tool and library called *gscope* that borrows some of its ideas from an oscilloscope.

Gscope provides a simple API that applications use to specify their "signals". Gscope actively monitors these software signals in real-time and displays them. Gscope can be used in this polling-driven manner or in a push-driven manner. When push-driven, applications send data to gscope and gscope displays this data passively. In this mode, Gscope can be used for correlation and visualization of distributed data.

Gscope simplifies visualization of low-latency software applications and has been used for visualizing time-dependent variables such as network bandwidth, latency, jitter, fill levels of buffers in a pipeline, CPU utilization, etc. In our experience, it has been an invaluable debugging and demonstration tool for the low-latency applications we have developed.

All of the components of our approach described above have been implemented and integrated in TSL, which has enabled us to evaluate how well these techniques meet the requirements of low-latency streaming applications. We use synthetic micro-benchmarks, simulated applications as well as real applications to evaluate these techniques. First each technique is evaluated in isolation and then these techniques are evaluated together under TSL. A network traffic generator is used to simulate a low-latency streaming application [63] and a media streaming application is used as a real low-latency application [64]. In each case, this work quantifies the latency incurred in the proposed system versus the current system. It shows that this metric can be improved significantly without degrading system throughput significantly.

1.3 Contributions of this Dissertation

This dissertation focuses on providing support for low-latency applications in general-purpose operating systems. It analyzes the sources of latency in an OS and presents several techniques that help reduce these latencies. The integration of these techniques allows streaming with latencies that are significantly lower than latencies in an unmodified general-purpose operating system. The

specific contributions of this dissertation are summarized below.

1. Design, implementation and evaluation of firm timers. Firm timers provide accurate timing with low overhead.

2. Integration and evaluation of different preemptible kernel schemes in a general-purpose OS.

3. Dynamic tuning of the size of the send buffer in the TCP stack, which significantly reduces output-buffer latency at a small expense in network throughput.

4. Design and implementation of a novel feedback-based CPU scheduling scheme that allows low-latency applications to easily express their timing constraints to the scheduler.

5. Design and implementation of a software oscilloscope that is an effective tool for visualizing and debugging low-latency software applications.

6. Integration of these techniques in a system called Time-Sensitive Linux (TSL).

7. Overall evaluation of TSL to show that it provides good support for real low-latency applications without significantly compromising the performance of throughput-oriented applications.

1.4 Outline of this Dissertation

The rest of this dissertation is organized as follows: Chapter 2 presents the design, implementation and evaluation of firm timers. Firm timers provide a low overhead and accurate timing mechanism that reduces timer latency. Chapter 3 describes different approaches that improve kernel responsiveness and experimentally evaluates each approach. Chapter 4 describes our adaptive buffer sizing mechanism that reduces output-buffering latency in TCP flows. The previous three chapters experimentally evaluate the performance and overheads of TSL and also present performance results for a real low-latency adaptive streaming application that has been developed in our research group. Chapter 5 describes various key real-time scheduling mechanisms including proportion-period scheduling. It presents the design and implementation of a feedback-based CPU scheduling scheme that allows inferring the CPU requirements of applications and dynamically assigning proportions and periods. Chapter 6 describes gscope, a visualization tool for low-latency

applications. Chapter 7 describes the related work in this field of research. Finally, Chapter 8 presents our conclusions on building system support for low-latency streaming applications. Also, it discusses several new research problems that emerge from this dissertation and the possible directions that can be explored to solve these problems.

Chapter 2

High-Resolution Timing

This chapter describes the design, implementation and evaluation of a high resolution timing mechanism called *firm timers* [38]. Our evaluation of firm timers shows that this mechanism significantly reduces the timer latency component of input latency and that it has low overhead.

2.1 Introduction

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

Traditionally, general-purpose operating systems have implemented their timing services 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 it 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. The following example explains the benefits of one-shot interrupts. Consider two threads 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.

2.2 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 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 [9]) 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 the use of periodic timers), it has now become inexpensive on modern hardware such as Intel Pentium II and later machines. For example, reprogramming the standard programmable interval timer (PIT) on an Intel x86 is very expensive because it requires several slow `out` instructions on the ISA bus. Each such instruction costs approximately one microsecond, which is 2000 cycles on a modern 2 GHz machine. 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 2.4.2.3, 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). Unlike with soft timers, where timer latency can be unbounded, firm timers limit timer latency to the overshoot value. Hence, the name firm timers.

With firm timers, 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.

Figure 2.1 shows that firm timers are programmed an overshoot amount of time after a time-sensitive application needs execution. The rectangular bars show the times when the kernel is executing, either in a system call or in some interrupt. At the end of each bar, the kernel checks for timer expiry. If the kernel executes between the time when the application needs execution and before the timer expires, then the cost of firm timers is simply the cost of reprogramming the timer.

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. The next section describes how our implementation can handle a mix of time-sensitive applications with differing

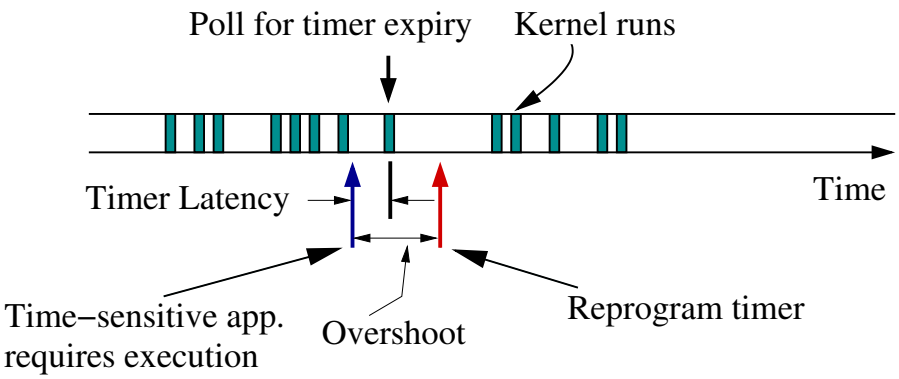


Figure 2.1: Overshoot in firm timers.

accuracy needs.

2.3 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 [16] 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.[1] 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.

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

[1]Note that soft timer checks occur at each interrupt and hence periodic timer interrupts that occur during the overshoot period fire pending but expired one-shot timers.

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 `select()`, `pause()`, `nanosleep()`, `setitimer()` 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 Chapter 4.7.

2.4 Firm Timers Evaluation

This section describes the experiments we performed to evaluate the behavior of firm timers. First, Section 2.4.1 presents experiments that quantify the timer latency of firm timers. Then Section 2.4.2 presents the performance overhead of the firm timers implementation on Time-Sensitive Linux as compared to the performance of timers on a standard Linux kernel.

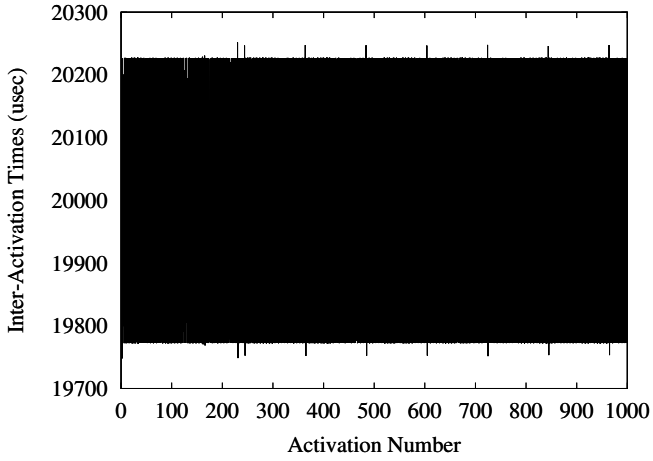
2.4.1 Timer Latency

Timer latency can be measured by using a typical periodic low-latency application. We implemented this application by running a process that sets up a periodic signal (using the `itimer()` system call) with a period T ranging from 100 μs to 100 ms. This process measures the time when it is woken up by the signal and then immediately returns to sleep. To measure this time, we read the Pentium Time Stamp Counter (TSC), a CPU register that is increased at every CPU clock cycle and can be accessed in a few cycles. Hence, the timing measurements introduce very low overhead and are very accurate. We calculated the difference between two successive process activations, which we call the *inter-activation time*. Note that in theory the inter-activation times should be equal to the period T. Hence, the deviation of the inter-activation times from T is a measure of timer latency. Since Linux ensures that a timer will never fire before the correct time, we expect this value to be 10 ms in a standard Linux kernel and to be close to the interrupt processing time with firm timers.

We run this program at the highest real-time priority to eliminate scheduling latency. Also, we

run these experiments on an idle system. In this case, few system calls will be invoked by other processes and a limited number of interrupts will fire and thus long non-preemptible execution paths or driver activations will not be triggered. However, note that we are unable to stop certain high-priority kernel processes such as the buffer-cache flush daemon from occasionally running during these experiments.

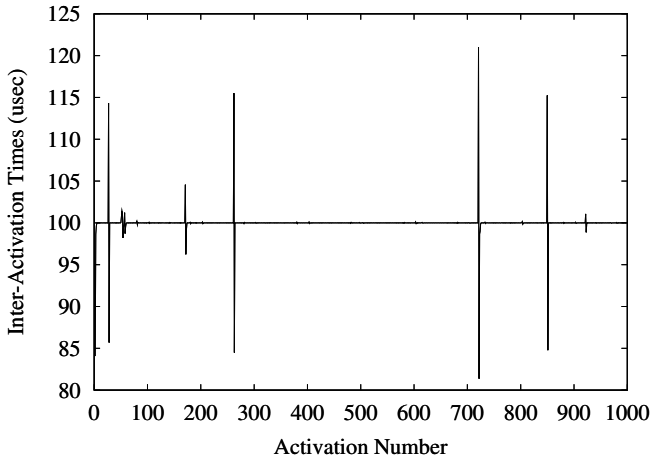
The experiments presented below were run on a 1.8 GHz Pentium processor with 512 MB of memory. Figure 2.2 shows the timer latency in standard Linux. In this experiment, the inter-activation times were measured when the period of the time-sensitive program is set to $T = 100$ μs. It shows that timer latency value can be larger than 10000 μs.[2] Figure 2.3 shows the same measurement on a firm timers kernel. This figure shows that the timer component of input latency can be easily removed by using firm timers. Note that after 1000 activations the maximum difference between the period and the actual inter-activation time is less that 25 μs.



The inter-activation time is the time between successive executions of a periodic process. The period of the process is 100 μs. Under Linux, we expect the inter-activation time to be close to 10 ms because of the timer granularity. Note that due to a bug in the Linux kernel, the observed inter-activation is closer to 20 ms.

Figure 2.2: Inter-activation times for a periodic thread with period 100 μs on standard Linux.

[2]We expect the timer latency to be about 10000 μs in the worst case on a lightly-loaded Linux system. We believe that the worst case in Figure 2.2, which is greater than 20000 μs, occurs as a result of a bug in the Linux timing code.



The inter-activation time is the time between successive executions of a periodic process. We expect the inter-activation time to be equal to the period of the process. In the figure above, each upward spike in the inter-activation time is followed by a downward spike because each wakeup is programmed with respect to an initial time and do not depend on the actual wakeup time.

Figure 2.3: Inter-activation times for a periodic thread with period 100 μs on TSL.

We repeated this experiment with different periods where each experiment was run for 10 million activations. These new experiments showed that the difference between the period and the inter-activation time does not significantly depend on the period T. Figure 2.4 shows the distribution of the inter-activation times when $T = 1000$ μs. This distribution does not significantly vary with increasing number of activations or decreasing period. Our data shows that the probability of inter-activation times being greater than 1015 μs or less than 985 μs is less than 0.01%. Note that the y-axis is on a log scale. The maximum measured inter-activation times is about 1300 μs, whereas the minimum is about 630 μs, but these large deviations are extremely rare (once in the 10 million activations).

We hypothesize that these large deviations are due to preemption latency caused by high-priority in-kernel threads over which we do not have execution control. Later, Chapter 3 presents more controlled experiments that show that latencies due to the various activities that trigger long non-preemptible sections can exceed 350 μs.

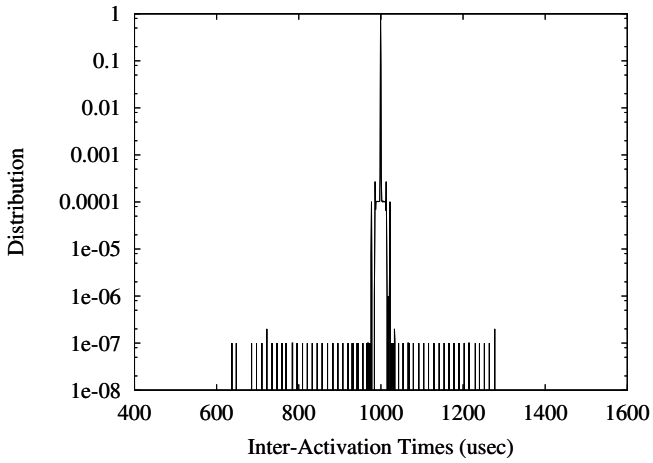


Figure 2.4: Distribution of inter-activation times when period is 1000 μs on TSL.

2.4.2 Overhead

Firm timers provide an accurate timing mechanism that 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 (see also Section 3.3.2). 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 in Section 2.4.2.3.

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. The time-sensitive process is the same as described in Section 2.4.1: a simple periodic thread that wakes up on a signal generated by a firm timer, measures the current time and then immediately goes to sleep each period. In the rest of this section, we refer to this thread 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 process 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.

2.4.2.1 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 2.5 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.

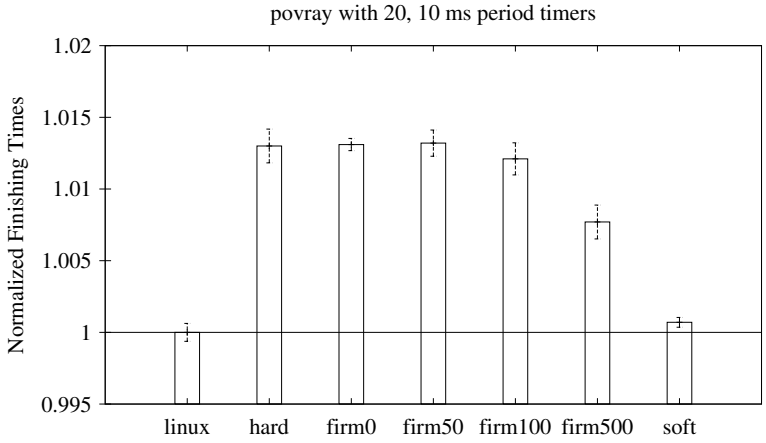


Figure 2.5: Overhead of firm timers in TSL with 20 timer processes.

Figure 2.6 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 pronounced 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 2.6, 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 addition, with 50 timers and a large 500 μs overshoot, soft timers fire often and thus firm timers have low overhead. In this case, the overhead of the scheduler on standard Linux dominates the overhead of the firm timers mechanism in TSL.

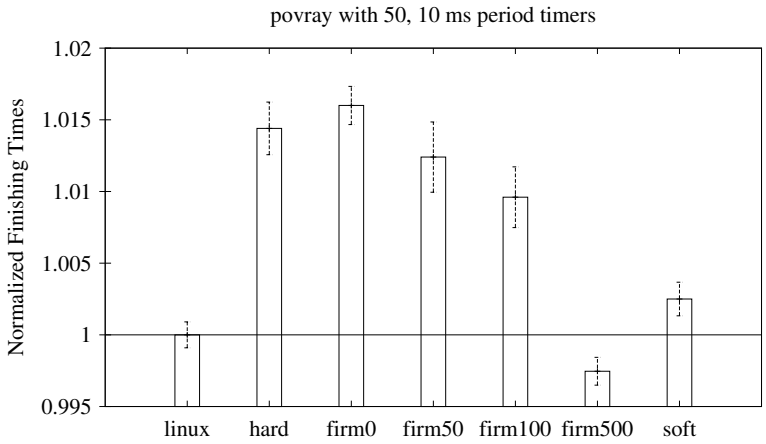


Figure 2.6: Overhead of firm timers in TSL with 50 timer processes.

2.4.2.2 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 2.7 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 2.5 and 2.6 is 0.5% and 1.6%.

2.4.2.3 Overhead Analysis

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

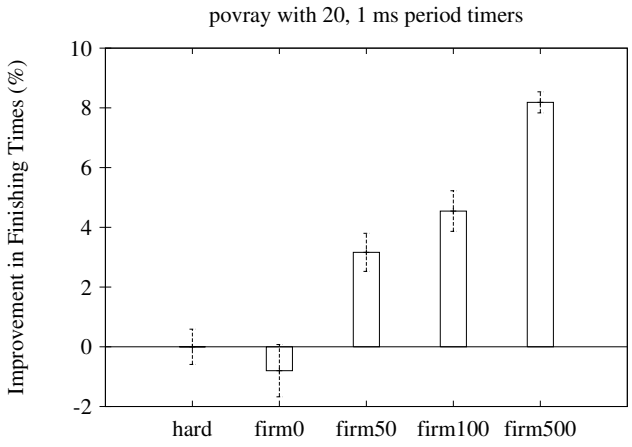


Figure 2.7: Comparison between hard and firm timers with different overshoot values on TSL.

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 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 Section 2.4.2. The total cost of firing firm timers is $C_cN_c + C_hN_h + C_sN_s$. If pure hard timers are used then the cost is C_hN_t. Hence, firm timers reduce the system overhead if

$$C_cN_c + C_hN_h + C_sN_s < C_hN_t$$

Rearranging the terms,

$$\begin{aligned} C_cN_c &< C_h(N_t - N_h) - C_sN_s \\ &< (C_h - C_s)N_s \end{aligned}$$

Rearranging the terms again,

$$N_s/N_c > C_c/(C_h - C_s) \tag{2.1}$$

Equation 2.1 shows that 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 $C_h = 8\ \mu$s, $C_s = 1\ \mu$s, and $C_c = 0.15\ \mu$s, 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 number 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 on the overshoot value. Aron and Druschel's original work on soft timers [9] 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.

2.5 Summary

Firm timers use one-shot timers to provide accurate timing to unmodified Linux applications in Time-Sensitive Linux. The timer latency of such timers is close to the interrupt service time. Firm timers use soft timers to reduce the cost of increased interrupts associated with fine-grained timing. In addition, they reduce the data structure overhead of one-shot timers by using periodic timers for large timeouts. Finally, firm timers allow making a trade-off between timer latency and overhead.

Chapter 3

Fine-Grained Kernel Preemptibility

This chapter reviews several techniques that improve kernel responsiveness and thus help to reduce the preemption latency component of input latency. Then we experimentally evaluate and compare these approaches and show that they reduce preemption latency significantly compared to a standard general-purpose OS such as Linux [2]. It also evaluates the overhead of these approaches and shows that they can be incorporated in general-purpose OSs without significantly affecting the performance of throughput-oriented applications [38].

3.1 Introduction

A kernel is more responsive when its non-preemptible sections that keep the scheduler from being invoked to schedule a thread are short. There are two main reasons why the scheduler may not be able to run when an interrupt is raised. One is that interrupts might be disabled. For example, if the timer interrupt in Figure 1.3 is disabled then the timer process 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 process, upon entering the ready queue, will be scheduled only when the other thread exits its non-preemptible critical section. These non-preemptible sections can be in the kernel or in kernel drivers. In addition, interrupt service routines (ISRs) and other kernel constructs such as bottom halves and tasklets in the Linux kernel, or deferred procedure calls (DPCs) in Windows, that execute on behalf of ISRs are also non-preemptible sections.

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. The simplest kernel structure disables 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). Thus, preemption 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.

Most current general-purpose OSs improve this structure by allowing preemption before certain long operations are invoked. For example, all versions of Linux and most other OSs such as Windows NT and Solaris allow preemption before invoking disk I/O operations. Unfortunately, even with this structure, Section 3.3 shows that preemption latency under standard Linux can be greater than 30 ms.

3.2 Improving Kernel Responsiveness

This section reviews three different approaches - *explicit preemption*, *preemptible kernels* and *preemptible lock-breaking kernels* - that change the structure of the kernel with the goal of improving preemption latency.

3.2.1 Explicit Preemption

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. This approach is a refinement of the current approach of allowing preemption only before "well-known" long kernel operations. Explicit preemption is used by some real-time OSs such as RED Linux [87] and by Morton's low-latency project [79] for Linux. Preemption latency in such a kernel decreases to the maximum time between two preemption points.

The choice of preemption points depends on execution flow paths in the kernel and these must be placed manually after careful auditing of system code under heavy loads, which helps determine long kernel paths. Preemption points are often placed in code that iterates over large or indefinite sized data structures such as linked lists. Essentially, a call to the scheduler is added, if scheduling is needed, after the iteration loop has crossed a certain threshold.

Preemption points can be easily placed in any function by simply adding calls to the scheduler

when data accessed by the function is private. However, if data structures are shared and the code holds a lock on it then the lock has to be dropped before the call to the scheduler and reacquired after the call to the scheduler. Otherwise, if a thread is preempted while holding a lock then deadlocks can occur. For example, on a uniprocessor, a second thread that tries to acquire the same lock will spin (assuming locks are implemented as spinlocks) and cause the system to deadlock because the first process will never have a chance to wake up. A compiler can help the process of preemption placement if it is able to detect that all locks have been released before and acquired after the call to the scheduler [29].

The releasing and acquiring of locks is also called *lock breaking*. Note that the lock can only be dropped if the kernel data structures can be placed in a consistent state. For example, if a list has to be processed atomically, where no other thread can modify the list while a function processes the list, then the lock cannot be broken easily. One option for handling this situation is to use a roll-back operation where the thread must abort or undo its operation on a lock release and restart its work from the beginning upon reacquiring the lock. This approach is often used in checkpointing systems [89], but it can be expensive because the undo operation requires saving state (possibly to disk) at each checkpoint [28].

Since lock breaking, in general, is like a roll-back operation, it has to be applied carefully in the kernel and requires a case-by-case analysis. In practice, lock breaking can often be applied without saving much state, and thus implemented efficiently, either because locks are held at a large granularity for efficiency rather than consistency, or because certain operations are idempotent. To understand these issues, consider the example shown in Figure 3.1. The function `process_list_function` in Figure 3.1 can execute in a non-preemptible section for an indefinite time since the list can be arbitrarily long. Note that the spinlock is held during the processing of the entire list rather than for each list member for two reasons: 1) to protect against changing the list, and 2) to avoid the overhead associated with keeping, acquiring and releasing locks for each list member.

Figure 3.2 shows `process_list_function` with an additional preemption point. This preemption point calls `schedule` after every 100 iterations of the loop that processes the list. The `schedule` function is only called if the `schedule_needed` variable returns true. This

```
int
process_list_function()
{
spinlock(&list_lock);
while (list) {
process_list_head(list);
list = list->next;
}
spinunlock(&list_lock);
}
```

Figure 3.1: An example of a function that processes a list in a long non-preemptible section.

variable is set by interrupts (or currently executing threads on other processors in an SMP machine) on behalf of threads that need to be scheduled. Note that before `schedule` is called, the lock `list_lock` must be released. This lock is reacquired after `schedule` returns to the function. After the lock is reacquired, some of the list members will be processed again because the function always starts at the head of the list, which requires the `process_list_head` function to be idempotent. To avoid redoing work, the `process_list_head` function can update the members of the list when it has already performed work on them in the past. For example, if this function flushes buffers to disk, then it can set and later test a bit in the list member that indicates whether the buffer is dirty.

The choice of the iteration threshold determines how much work is done between preemption points. A small threshold is desirable because it decreases the time between preemption points. However, a small threshold can increase overhead because the spinlock that would otherwise have been held may have to be repeatedly released and reacquired. In addition, under heavy load, when several threads are runnable and need to be scheduled (`schedule_needed` in Figure 3.2 is set to true), a small threshold can cause a form of thrashing where the system does not perform much useful work. For example, if the threshold is set to one, then the function in Figure 3.2 may continuously loop between the "`redo:`" and the "`goto redo`" regions without ever processing the list.

```
int
process_list_preemptible_function()
{
    int count = 0;                                                    |
                                                                      |
redo:                                                                 |
    spinlock(&list_lock);
    while (list) {
        if (count++ < 100) { /* threshold is 100 */                   |
            count = 0;                                                |
            if (schedule_needed) {                                    |
                spinunlock(&list_lock);                               |
                schedule();                                           |
                goto redo;                                            |
            }                                                         |
        }                                                             |
        process_list_head(list);
        list = list->next;
    }
    spinunlock(&list_lock);
}
```

The bars on the right show the additional code that has been added to the list processing function shown in Figure 3.1.

Figure 3.2: The list processing function with an explicit preemption point.

The process of finding large non-preemptible code sections, adding preemption points at appropriate points in this code and choosing the appropriate iteration threshold can be a very time-intensive process. In addition, it has to be a continuing process as the kernel code evolves over time. One approach, used by Morton [79], that helps to find large non-preemptible sections is to use a periodic timer that detects whether the scheduler has been invoked between two timer expirations. If the scheduler has not been invoked then the stack backtrace at the second timer expiration gives an indication of the routines that have a large non-preemptible section.

3.2.2 Preemptible Kernels

Another approach, used in many real-time systems, is to allow preemption any time the kernel is not accessing shared kernel data. To support this fine level of kernel preemptibility, all shared

kernel data must be explicitly protected using mutexes or spinlocks. The Linux preemptible kernel project [68] uses this approach and only disables kernel preemption on a processor when an interrupt handler is executing or a spinlock is held on that processor. Preemption is disabled for interrupt handlers because they are typically short sections of code that assume non-preemptible execution and do not lock their data at a fine-granularity for efficiency. Preemption is disabled when spinlocks are held for two reasons. First, on a uniprocessor, as explained earlier, if a thread is preempted while holding a spinlock then another thread can deadlock the system if it tries to acquire the same spinlock. Second, spinlocks are assumed to be held for short periods and preempting a thread that holds a spinlock can cause extreme degradation in system performance when the spinlock is held on a resource that is heavily accessed. For example, a thread that holds a spinlock on a shared, global run queue can effectively disable scheduling on all of the processors in an SMP machine because the other processors, when running `schedule`, would spin on the run queue lock while the thread is preempted.

The kernel checks whether any spinlock is held in the current processor whenever a spinlock is released or at the end of an interrupt handler. If no spinlocks are being held and if scheduling is needed, then the kernel calls `schedule`, similar to the code shown in Figure 3.2. In a preemptible kernel, the preemption latency is determined by the sum of the maximum amount of time for which a spinlock is held inside the kernel and the maximum time taken by interrupt service routines.

3.2.3 Preemptible Lock-Breaking Kernels

Preemption latency can be high in preemptible kernels when spinlocks are held for a long time. Lock breaking addresses this problem by "breaking" long spinlock sections (i.e., by releasing spinlocks at strategic points within code that accesses shared data structures with spinlocks). This approach is a combination of the explicit preemption and the preemptible kernel approaches. Preemptible lock-breaking kernels reduce the size of non-preemptible sections, but do not decrease the amount of non-preemptible time spent in interrupt service routines.

3.3 Evaluation

The explicit preemption approach has been implemented for the Linux kernel by Morton [79]. The preemptible kernel design has been implemented for Linux by Love [68]. Love has also integrated Morton's preemption code in his preemptible kernel to produce the preemptible lock-breaking kernel.

While Morton and Love have done some preliminary evaluation of their approaches, there has been no systematic evaluation that compares these approaches. In this thesis, our goal is to evaluate and compare these approaches to determine their effectiveness at reducing preemption latency. Our expectation is that both these approaches will significantly reduce preemption latency compared to that in standard Linux. In addition, we expect that the third approach which combines the first two approaches will yield the best results. After doing this evaluation, we plan to choose the best approach and integrate it as part of our TSL system.

For our evaluation, we use 1) the standard *Linux* kernel, 2) the *Low-Latency Linux* kernel from Morton that uses explicit preemption points, 3) the *Preemptible Linux* kernel and 4) the *Preemptible Lock-Breaking Linux* kernel, both from Love. The evaluations were performed on version 2.4.20 of all the Linux kernels. In Section 3.3.1, our evaluation measures the maximum preemption latency in each of these kernels under heavy system load and shows that the latter three variants improve preemption latency significantly as compared to standard Linux. Then, Section 3.3.2 measures the overheads of these approaches and shows that they have a small impact on the behavior of throughput-oriented applications and are thus suitable for general-purpose OSs.

3.3.1 Preemption Latency

We measure preemption latency by using a process that invokes sleep for a specified amount of time (using the `usleep` system call) and then measures the time that it actually slept. In general, the difference between these two times is input latency. To measure preemption latency, the timer latency and scheduling latency components must be removed from input latency. Section 2.4.1 showed that firm timers have timer latencies that are very small and close to the interrupt service times. Hence, we use firm timers in the `usleep` implementation to eliminate timer latency. To eliminate scheduling latency, we run the `usleep` test program at the highest real-time priority.

To measure preemption latency, two times must be measured: time t_1 before going to sleep and time t_2 after being woken up. Suppose the process needed to sleep for time T. Then the preemption latency is $t_2 - (t_1 + T)$. Times t_1 and t_2 are read using the Pentium Time Stamp Counter (TSC), a CPU register that is increased at every CPU clock cycle and can be accessed in a few cycles. Hence, the measurements have low overhead and are very accurate.

We investigated how various system activities contribute to preemption latency by running various background applications. The following applications shown below are known to invoke long system calls or cause frequent interrupts and thus they trigger long non-preemptible sections either in the Linux kernel or in the drivers. Experience and careful code analysis by various members of the Linux community (for example, see Senoner [100]) confirms that the list of latency sources shown below is comprehensive (i.e., it triggers a representative set of long non-preemptible sections in the Linux kernel and in the Linux drivers).

Memory Stress: One potential way to increase input latency involves accessing large amounts of memory so that several page faults are generated in succession. The kernel invokes the page fault handler repeatedly and can thus execute long non-preemptible code sections.

Caps-Lock Stress: A quick inspection of the kernel code reveals that, when the num-lock or caps-lock LED is switched, the keyboard driver sends a command to the keyboard controller and then (perhaps for simplicity) spins while waiting for an acknowledgment interrupt. This process can potentially disable preemption for a long time.

Console-Switch Stress: The console driver code also seems to contain long non-preemptible paths that are triggered when switching virtual consoles.

I/O Stress: When the kernel or the drivers have to transfer chunks of data, they generally move this data inside non-preemptible sections. Hence, system calls that move large amounts of data from user space to kernel space (and vice-versa) and from kernel memory to a hardware peripheral, such as the disk, can cause large latencies.

Procfs Stress: Other potential latency problems in Linux are caused by the `/proc` file system. The `/proc` file system is a pseudo file system used by Linux to share data between the kernel and user programs. Concurrent accesses to the shared data structures in the `proc`

file system must be protected by non-preemptible sections. Hence, we expect that reading large amounts of data from the `/proc` file system can increase preemption latency.

Fork Stress: The `fork()` system call can generate high latencies for two reasons. First, the new process is created inside a non-preemptible section and involves copying large amounts of data including page tables. Second, the overhead of the scheduler increases with increasing number of active processes in the system.

In our experiments, we used the `usleep()` test program described above with $T = 100$ μs to measure preemption latency. This program is started on an unloaded machine in single user mode to have better control over the system activities running in the background. Then the load-generating applications described above are run in the background to trigger long non-preemptible paths. To easily represent the latency results in a single plot per kernel, we used a background load that was generated as follows:

1. The memory stress test allocates a large integer array with a total size of 128 MB and then accesses it sequentially one time. This test starts at 1000 ms, and finishes around 2000 ms. No background process runs from 2000 ms to 7000 ms.

2. The caps-lock stress test runs a program that switches the caps-lock LED twice. This test turns on the LED at 7000 ms and then turns it off at 8000 ms.

3. The console-switch stress test runs a program that switches virtual consoles on Linux twice, first at 9000 ms and then at 10000 ms.

4. The I/O stress test opens an 8 MB file and then reads and writes from this file 2 MB at a time. The reads and writes are done repeatedly five times. This test starts at 11000 ms and finishes around 13000 ms.

5. The procfs stress test reads a 512 MB file in the `/proc` file system. It runs from 17000 ms to around 18000 ms.

6. The fork test forks 512 processes. This test starts at 20000 ms.

The times at which each of these experiments were run is shown later in Figure 3.3 and in Figure 3.4. We performed these experiments on the standard Linux kernel, the Low-Latency Linux kernel, the Preemptible Linux kernel, and the Preemptible Lock-Breaking Linux kernel. The next section describes the initial set of experiments we performed to understand which activities cause large preemption latencies. Section 3.3.1.2 describes additional experiments that we performed to test the sensitivity of the system to the order and the length of experimental runs.

3.3.1.1 Initial Analysis

Table 3.1 shows the results of running our experiments. It shows that the Low-Latency kernel reduces preemption latency during the memory stress test and the I/O stress test, but does not reduce latency due to the console switch, the caps-lock switch, and the procfs stress test. On the other hand, the Preemptible kernel reduces the latency due to the procfs stress test, but reintroduces a large latency during the memory stress test. Finally, the Lock-Breaking kernel seems to provide the benefits of the Low-Latency kernel (the latency during the memory stress test is low) together with the benefits of the Preemptible kernel (for instance, the latency is reduced in the console switch test and in the procfs stress test). In this case, the largest preemption latency is caused by the caps-lock stress test but all other latencies are within 1 ms.

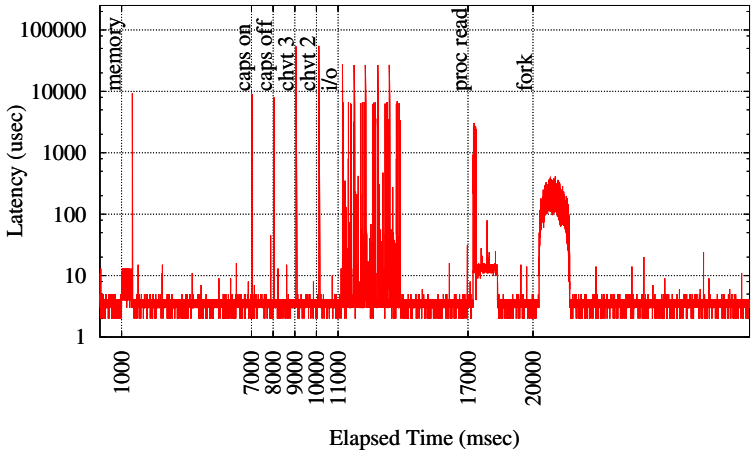
	Memory Stress	Caps-Lock Stress	Console Switch	I/O Stress	Procfs Stress	Fork Stress
Standard Linux	18212 μs	6487 μs	614 μs	27596 μs	3084 μs	295 μs
Low-Latency	63 μs	6831 μs	686 μs	38 μs	2904 μs	332 μs
Preemptible	17467 μs	6912 μs	213 μs	187 μs	31 μs	329 μs
Pre. Lock-Breaking	54 μs	6525 μs	207 μs	162μs	24 μs	314 μs

These tests were run for 25 seconds. The figures shown are the worst-case (maximum) numbers.

Table 3.1: Preemption latencies for four different kernels under different loads.

Figure 3.3 graphically shows the results for the standard Linux kernel together with firm timers and Figure 3.4 shows the results of the same experiments on a Preemptible Lock-Breaking kernel. These graphs provide further insight into the causes of preemption latency (for the sake of brevity, we omit the plots for other kernels, which are similar to these graphs). For instance, Figure 3.3

shows that the large latency in the memory stress test that we see in Table 3.1 occurs only at the termination of the program. We found that the source of this latency is the `munmap()` system call which unmaps large memory buffers during program exit.



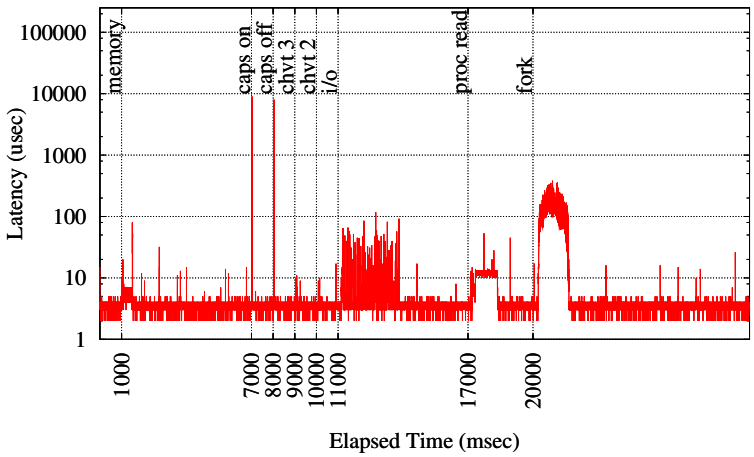
This test is performed under heavy background load. It uses firm timers to remove the effect of timer latency. and The maximum preemption latency exceeds 25 ms.

Figure 3.3: Preemption latency on a Linux kernel.

Figures 3.3 and 3.4 also show the resolution of firm timers. When the system is not loaded, such as after the fork test, the latency lies between 2-5 μs. This is the latency from the time that the hard timer fires to the time when the application is activated and gets control. Hence, firm timers can be programmed with this fine-granularity. Of course, programming at this granularity will cause the system to spend almost all its time in the kernel. However, these numbers help in estimating the overhead of using fine-grained timing with firm timers. For example, we expect system overhead due to firm timers to be between 10-25% when firm timers are programmed at 20 μs granularity on a 1.8 GHz processor.

3.3.1.2 Sensitivity Analysis

For sensitivity analysis, we performed additional experiments by running the stress test programs in several different orders and for different lengths of time. Table 3.2 shows the maximum OS latency measured when running the memory stress test, the I/O stress test, the procfs stress test



This test is performed with firm timers and under heavy background load. Note that all latencies (except the caps-lock test latency at 7000 and 8000 ms) are under 1 ms.

Figure 3.4: Preemption latency on a Preemptible Lock-Breaking Linux kernel.

and the fork stress test for a long time (the tests were run for 10 hours and 36 million samples were collected).

	Memory Stress	I/O Stress	ProcFS Stress	Fork Stress
Standard Linux	18956 μs	28314 μs	3563 μs	617 μs
Low-Latency	293 μs	292 μs	3379 μs	596 μs
Preemptible	18848 μs	392 μs	224 μs	645 μs
Pre. Lock-Breaking	239 μs	322 μs	231 μs	537 μs

These tests were run for 10 hours. The figures shown are the worst-case (maximum) numbers.

Table 3.2: Maximum preemption latencies for four different kernels under different loads.

We do not show the console switch and caps-lock tests results because they did not show any difference with respect to the values in Table 3.1. These experiments confirmed that none of the evaluated patches reduces the caps-lock switch latency. In addition, the Preemptible and Lock-Breaking kernels significantly reduce the console switch latency with respect to the standard or the Low-Latency kernel.

Although the worst case values shown in Table 3.2 are higher than in Table 3.1, the results

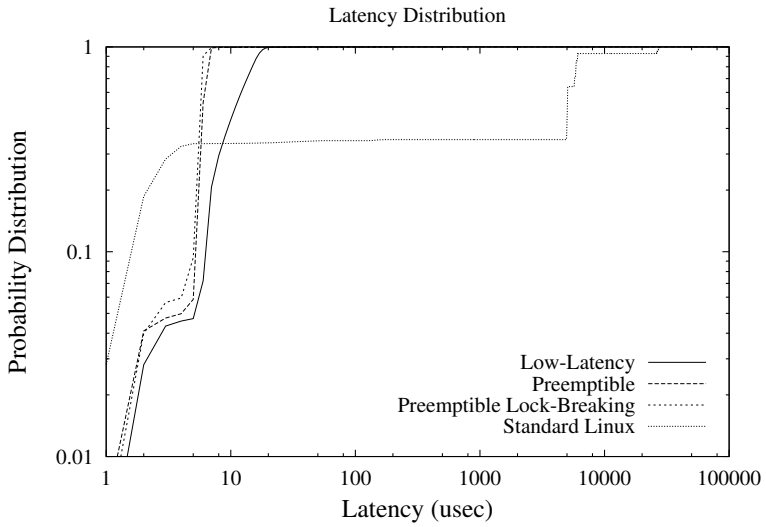
are qualitatively similar. Thus, 1) the Low-Latency kernel reduces latency during the memory stress test and the I/O stress test but not during the procfs stress test or during console switch tests (not shown here), 2) the preemptible kernel reduces latency during the procfs stress test and during the console switch tests (not shown here) but not during the memory stress test, and 3) the Lock-Breaking kernel reduces all these latencies.

In addition to the worst case latencies, we are also interested in looking for outlier latencies for which we plot a distribution of the preemption latencies as shown in Figure 3.5. This figure shows the latencies measured during the I/O stress test in the standard Linux kernel, the Low-Latency kernel, the Preemptible kernel and the Lock-Breaking kernel. Note that for the latter three kernels the probability of measuring latencies higher than 20 μs is less than 0.01. The graph shows that the Preemptible and Lock-Breaking kernels have lower latency with higher probability (their probability distribution rises faster). For example, the probability of having latencies larger than 10 μs is 0.00534 on a Preemptible kernel and 0.00459 on a Lock-Breaking kernel but 0.558 on a Low-Latency kernel). Hence, we see that the Lock-Breaking kernel performs slightly better than the Low-Latency and the Preemptible kernels. Based on these results, we have integrated Love's Lock-Breaking kernel patch into TSL. In the rest of the thesis, when we perform experiments on TSL, we use this kernel preemption code.

3.3.1.3 Discussion

We noticed in Table 3.2 that the Low-Latency kernel has the smallest worst-case latency for the I/O stress test even though its probability distribution function in Figure 3.5 is poorer than the Preemptible and the Lock-Breaking kernels. To understand this issue further, we zoomed in on the top part of Figure 3.5 which would help us see the worst case better.

Figure 3.6, which presents the zoomed plot, shows the Preemptible and the Lock-Breaking kernels have latency distributions with longer tails in 0.2% cases. For example, the probability of having latencies larger than 40 μs is $290.1 * 10^{-5}$ on a Preemptible kernel, $183.5 * 10^{-5}$ on a Lock-Breaking kernel but only $12.6 * 10^{-5}$ on a Low-Latency kernel. Hence, the Preemptible and the Lock-Breaking kernels perform better if latencies less than 20 us are needed while the Low-Latency kernel is slightly better (in 0.2% cases) if latencies between 20 us and 100 us are needed. We are currently investigating why the Lock-Breaking kernel performs worse than the



This test is performed with firm timers and with the I/O stress test in background. The Preemptible and Preemptible Lock-Breaking kernels have lower latency with higher probability as compared to the Low-Latency kernel because their probability distribution rises faster.

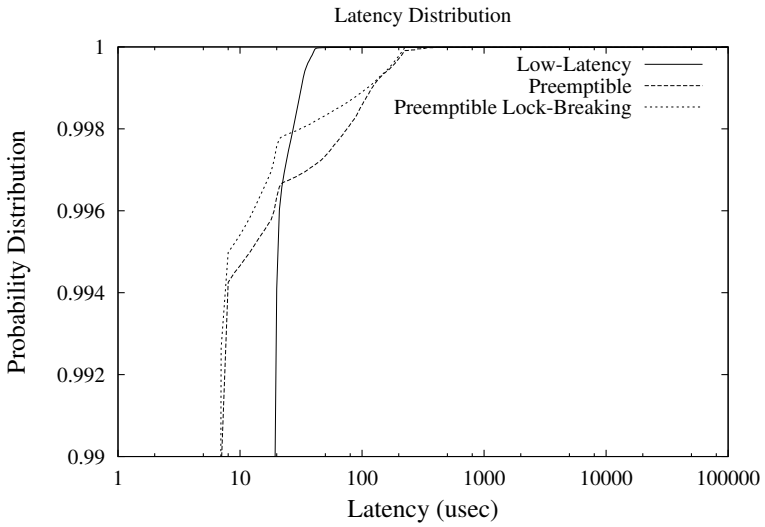
Figure 3.5: Distribution of latency on different versions of the Linux kernel.

Low-Latency kernel in this region even though it has all the preemption points that exist in the Low-Latency kernel. One reason may be that the preemptible nature of the Lock-Breaking kernel exposes some long-running paths that were not possible or highly improbable in the Low-Latency kernel.

3.3.2 Overhead

The overhead of the explicit preemption and the preemptible kernel approaches lies in the cost of executing code at the newly inserted preemption points. 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 cost for executing preemption because that depends on the workload. In particular, one instance where we expect that more preemption will occur with these approaches is when firm timers are used, since firm timers can cause preemption at a finer granularity. The overhead of executing preemption is taken into account in our firm-timer overhead



This figure shows that although the Low-Latency kernel has higher latency than the Preemptible and Preemptible Lock-Breaking kernels 99.8% times, it has a shorter tail and thus has better worst-case performance.

Figure 3.6: A zoom of the top part of Figure 3.5.

experiments in Section 2.4.2, where we showed that firm timers have no more than 1.5% overhead as compared to timers in standard Linux. Hence, the overhead of executing preemption in this case must be less than 1.5%.

Here we measure the cost associated with checking for preemption in TSL. Our TSL implementation incorporates the Lock-Breaking Preemptible kernel code which combines both the preemptible kernel and the explicit preemption approaches. Hence, our measurements should give a worst case overhead number (i.e., the overhead should be greater than either of these approaches).

We measure the cost of checking for preemption by running the set of benchmarks described in Section 3.3.1 that are known to stress preemption latency in Linux. In particular, we ran the memory access test, the fork test and the 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 much 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 copies data from a user

buffer to a file that is 8 MB long and flushes the buffer cache. It also reads data from an 8 MB long file. The reads are writes are done several times, 2 MB at a time. 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 execution 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 versus standard Linux is very low.

3.4 Application-Level Evaluation

This section evaluates the timing behavior of a low-latency application running on Time-Sensitive Linux. Until now, we have evaluated the timer and preemption latencies in the kernel in isolation through micro-benchmarks. Here, we perform evaluation at the application level and show how these techniques help to improve the performance of a realistic low-latency application called `mplayer` [80]. Mplayer is a stored multimedia player that can handle several different media formats. We measure the improvement in audio/video synchronization in mplayer under Time-Sensitive Linux compared to standard Linux.

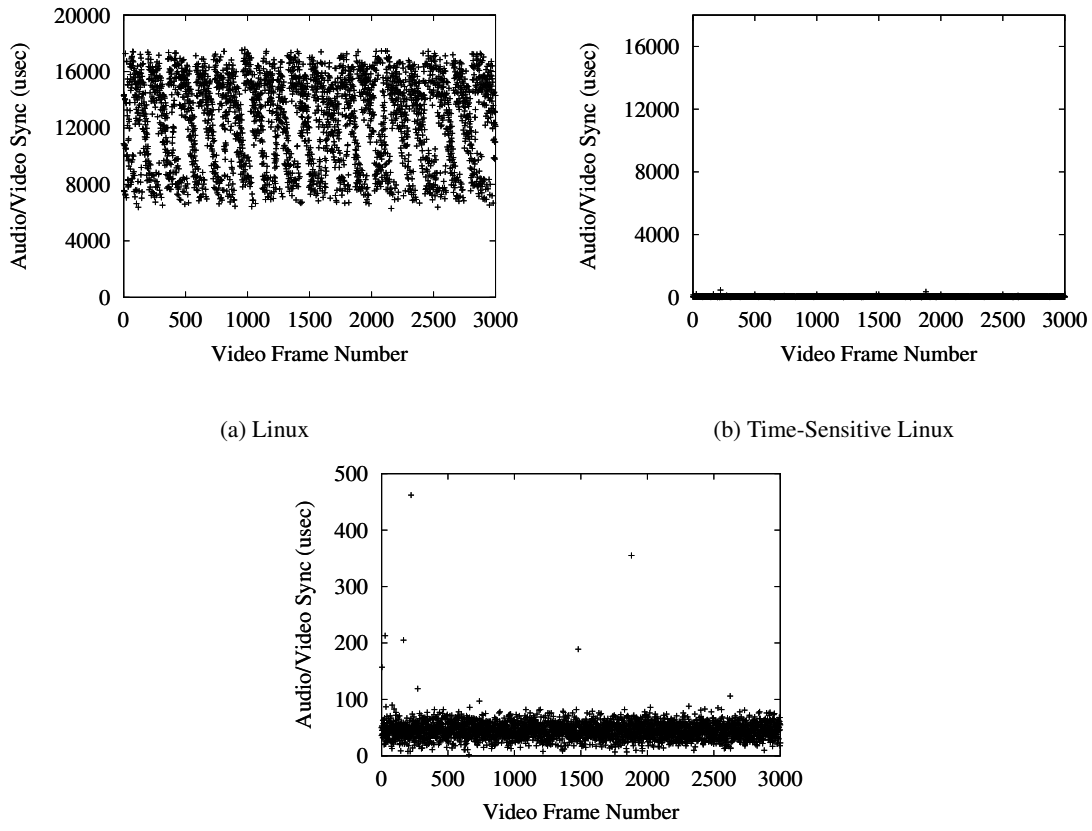
Mplayer synchronizes audio and video streams by using time-stamps that are associated with the audio and video frames. The audio card is used as a timing source (i.e., ideally the video frame is displayed when its time-stamp is the same as the time-stamp of the currently playing audio frame). To do so, audio samples are written into the audio card buffer, 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 (i.e., a video deadline has been missed) and it is immediately displayed. Otherwise, the system sleeps until the difference between the video and audio time-stamps 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. 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 hence we only present results for audio/video synchronization skew. To avoid a temporary overload situation, where a frame cannot be decoded on time, we use a small sized video clip that takes less than 20% of the CPU on an average.

We compare the audio/video skew of mplayer on standard Linux and on 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 the 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. We expect disk activity to effect the timing of mplayer under both Linux and TSL. However, TSL has shorter non-preemptible sections and hence the mplayer timing should not be affected much. In each of these tests, mplayer is run for 100 seconds at real-time priority. To avoid priority inversion effects, the X11 server is also run at the same real-time priority as mplayer.

3.4.1 Non-kernel CPU Competing Load

Figure 3.7 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 but is run at a lower priority than mplayer and the X11 server. Figure 3.7 (a) shows that for standard Linux the maximum skew is large and close to 18000 μs while Figures 3.7 (b) and 3.7 (c) shows that the skew on TSL is significantly smaller and is less than 500 μs. This result can be explained by the fact that the system is relatively unloaded in terms of kernel activity (i.e., the CPU stress test runs user-level code), and the skew on standard Linux in this case is dominated by timer latency which can be as large as 22000 μs (see Section 2.4.1). On TSL, timer latency is small (less than 2-5 μs) and hence the latency is dominated by preemption latency, which can be



(a) Linux

(b) Time-Sensitive Linux

(c) Time-Sensitive Linux (zoom)

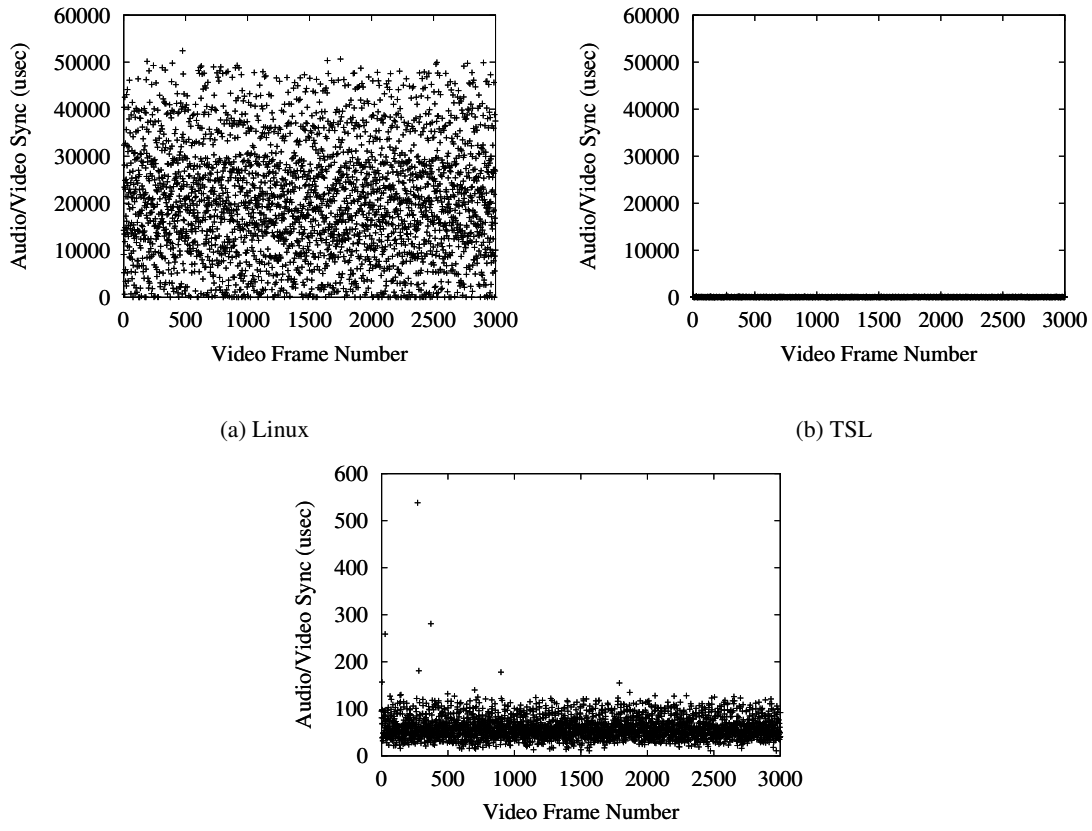
Background load is a user-level CPU stress test that runs an empty loop. The mplayer process and the X server are run at real-time priority. Figure (c) is a zoom of Figure (b).

Figure 3.7: Audio/video skew on Linux and TSL under non-kernel CPU load.

caused for various reasons such as reading the MPEG file from disk, etc. (see Section 3.3.1).

3.4.2 Kernel CPU Competing Load

The second experiment compares the audio/video skew in mplayer on Linux and on TSL when the background load copies a large 8 MB memory buffer to a file with a single `write` system call. Figure 3.8 (a) shows the audio/video skew is as large as 60000 μs for Linux. In this case, the kernel moves the data from the user to the kernel space in a non-preemptible section. Figures 3.8 (b) and 3.8 (c) shows that the maximum skew is less than 600 μs on TSL. Linux. This improvement occurs



(a) Linux

(b) TSL

(c) TSL (zoom)

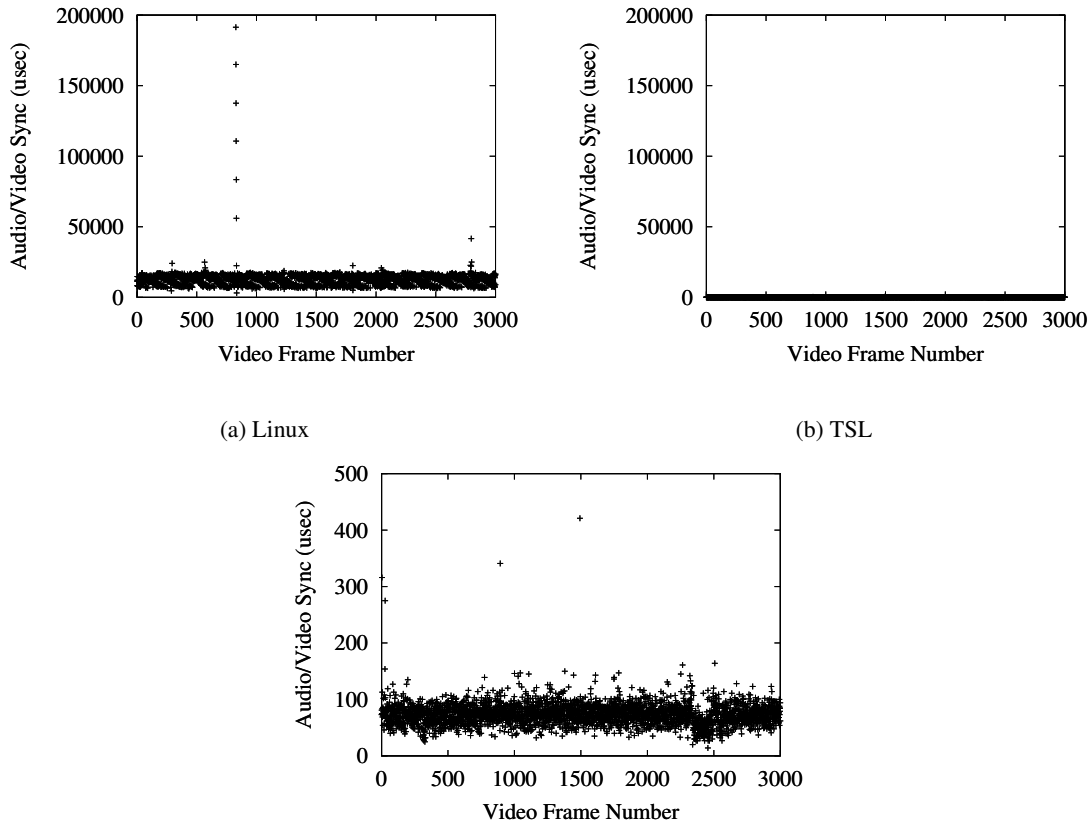
Background load copies a 8 MB buffer from user level to a file with a single `write` call. Figure (c) is a zoom of Figure (b).

Figure 3.8: Audio/video skew on Linux and on TSL with kernel CPU load.

as a result of improved kernel preemptibility for large write calls in TSL.

3.4.3 File System Competing Load

The third experiment compares the audio/video skew in mplayer on Linux and on 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 3.9 (a) shows that the skew under Linux can be as high as 200000 μs while Figures 3.9 (b) and 3.9 (c) show that



(a) Linux

(b) TSL

(c) TSL (zoom)

Background load repeatedly copies a compiled Linux kernel sources directory recursively and then flushes the file system. Figure (c) is a zoom of Figure (b).

Figure 3.9: Audio/video skew on Linux and on TSL with file-system load.

skew is less than 450 μs on TSL. This result shows that TSL can provide low latencies even under heavy file-system and disk load.

3.5 Conclusions and Future Work

In this chapter, we have reviewed various schemes that have been proposed to reduce preemption latency in general-purpose OSs. Our goal was to evaluate these schemes, which have been implemented by others in the context of the Linux kernel, to determine how well they reduce preemption

latency in the presence of heavy system loads. After this evaluation, we wanted to incorporate the best scheme is TSL. Our evaluation showed, a little surprisingly, that a kernel in which preemption points have been explicitly placed based on kernel profiling can often have lower preemption latency than the preemptible kernel approach. However, as expected, combining these two approaches generally yields the best results. As a result, we have incorporated the Lock-Breaking Preemption code in TSL. Our overhead experiments show that the additional cost of checking for preemption in TSL compared to Linux is very low.

In the future, we plan to compare the performance of the Low-Latency kernel and the Lock-Breaking kernel in more detail as explained in Section 3.3.1.3.

Recall from Section 2.2, which described the design of firm timers, that firm timers use soft timers to deal with the problem of interrupt overhead associated with fine-grained timing. 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.

Chapter 4

Adaptive Send-Buffer Tuning

In this chapter, we develop an adaptive buffer-size tuning technique that reduces output latency for TCP, the most commonly used transport protocol on the Internet today. This technique enables low-latency video streaming over TCP. TCP-based video streaming is desirable because TCP provides congestion controlled delivery, which is largely responsible for the remarkable stability of the Internet despite an explosive growth in traffic, topology and applications [53]. In addition, TCP handles flow control and packet losses, so applications do not have to perform packet loss recovery. This issue is especially important because the effects of packet loss can quickly become severe. For instance, loss of the header bits of a picture typically renders the whole picture and possibly a large segment of surrounding video data unviewable. Thus media applications over a lossy transport protocol have to implement complex recovery strategies such as FEC [96] that potentially have high bandwidth and processing overhead.

The buffer tuning technique does not change the TCP protocol and is thus attractive in terms of deployment. We implement this technique in TSL and our evaluation shows that a significant portion of end-to-end latency in TCP flows occurs as a result of output-buffer latency and that adaptive buffer sizing is able to remove this latency component [39]. Hence, adaptive buffer tuning enables low-latency streaming applications, such as responsive media control operations (e.g., the sequence of start play, fast forward and restart play) and video conferencing, over the TCP protocol.

Note that adaptive send-buffer size tuning reduces output-buffer latency in TCP, but introduces this same latency at the application level because applications are only allowed to write packet data at a later point in time. Fortunately, this issue is not a problem because low-latency streaming applications can adapt their bandwidth requirements using techniques such as prioritized data

dropping and dynamic rate shaping [94, 31, 64]. For example, if TCP does not allow timely writes, the sender application can drop low-priority data and then send timely and high priority data at the next write, which will arrive at the receiver with low delay. With low output-buffer latency, a quality-adaptive application can delay committing data to TCP and thus it can send more timely data.

While our technique reduces latency, it also reduces network throughput. We explore the reasons for this effect and then propose a simple enhancement to the technique that allows trading latency and network throughput.

4.1 Adapting Send-Buffer Size

Recall that output latency is the time between when the application generates output to the time when this output is delivered to the external world. For TCP flows, this is the time from when an application writes to the kernel to the time that data is transmitted on the network interface on the sender side. To understand the reasons for output latency in TCP, we need to briefly examine TCP's transmission behavior. TCP is a window-based protocol, where its window size is the maximum number of unacknowledged and distinct packets in flight in the network at any time. TCP stores the size of this current window in the variable CWND. When an acknowledgment (or ACK) arrives for the first packet that was transmitted in the current window, the window is said to have opened up, and then TCP transmits a new packet. Given a network round-trip time RTT, the throughput of a TCP flow is roughly CWND/RTT, because CWND packets are sent by TCP every round-trip time.

Since TCP is normally used in a best-effort network, such as the Internet, it must estimate bandwidth availability. To do so, it probes for additional bandwidth by slowly increasing CWND, and hence its transmission rate, by one packet every round trip time, as shown in Figure 4.1. Eventually, the network drops a packet for this TCP flow, and TCP perceives this event as a *congestion event*. At these events, TCP drops its CWND value by half to reduce its transmission rate.

Before TCP transmits application packets, it stores them in a *fixed size* send buffer, as shown in Figure 4.2. This buffer serves two functions. First, it handles rate mismatches between the

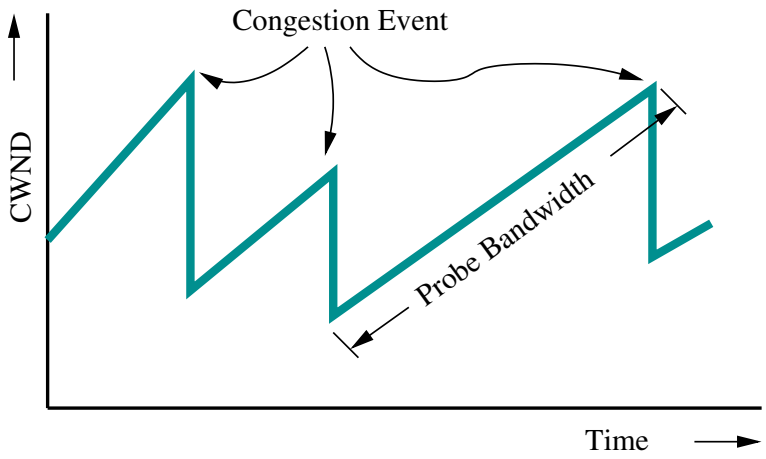


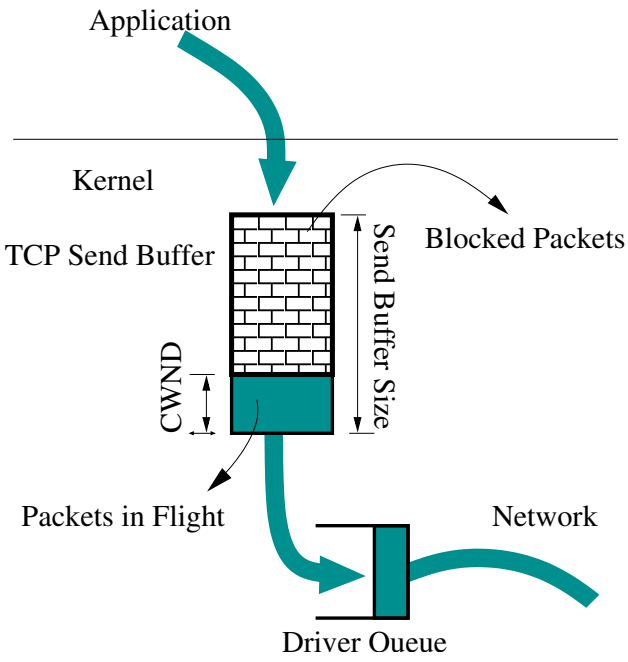
Figure 4.1: TCP congestion window (CWND).

application's sending rate and TCP's transmission rate. Second, since TCP is a reliable protocol, this buffer is used to keep copies of the packets in flight (its current window) so they can be retransmitted if needed. Since CWND stores the number of packets in flight, its value can never exceed the send buffer size. Hence, a small size send buffer can significantly restrict throughput since TCP throughput is proportional to CWND.

Unfortunately, if a fixed size send buffer is large, it can introduce significant output latency in a TCP stream. Consider the following example of how it affects latency. The send buffer in most current Unix kernels is at least 64 KB. For a 300 Kbs video stream, a full send buffer contributes 1700 ms of delay. This delay increases for a smaller bandwidth flow or when the flow faces increasing competition since the stream bandwidth goes down. By comparison, the round trip time generally lies between 50-100 ms for coast-to-coast transmission within the United States.

Figure 4.2 shows that the first CWND packets in the send buffer have been transmitted or are in flight. Hence, these packets do *not* contribute to any output latency for a packet that is newly accepted in the send buffer. However, the rest of the packets have to sit in the send buffer until acknowledgments have been received for the previous packets. We refer to these packets as blocked packets. Unfortunately, these packets contribute to output latency for a TCP flow.

It should be clear from this discussion that output latency can be minimized if the size of the send buffer is no more than CWND packets because a larger buffer allows blocked packets and



TCP's send buffer serves two functions. It matches the application's rate with the network transmission rate. It also stores CWND packets that are currently being transmitted in case they are needed for retransmission. These packets do not introduce any output latency because they have already been transmitted. The rest of the packets (shown as blocked packets in the figure) can add significant output latency.

Figure 4.2: TCP's send buffer.

thus introduces output latency. However, a buffer smaller than CWND packets is guaranteed to limit throughput because CWND gets artificially limited by the buffer size rather than congestion or receiver buffer size feedback in TCP. Hence, tuning the send buffer size to follow CWND should minimize output latency without significantly affecting flow throughput. Since CWND changes dynamically over time, as shown in Figure 4.1, we call this approach adaptive send-buffer tuning.

In essence, this approach separates the two functions of the send buffer: holding packets for retransmission and rate matching. The first function doesn't add output latency. The second can add significant output latency and should be eliminated for low-latency streams. We have implemented this approach, which we call MIN_BUF TCP, in Time-Sensitive Linux (TSL). Our implementation is described in Section 4.4.

A MIN_BUF TCP flow blocks an application from sending data when there are CWND packets in the send buffer. Later, the application is allowed to send data when at least one packet can be admitted in the send buffer. Consider the operation of a MIN_BUF TCP flow. The send buffer will have at most CWND packets after an application writes a packet to the socket. MIN_BUF TCP can immediately transmit this packet since this packet lies within TCP's window. After this transmission, MIN_BUF TCP will wait for the arrival of an ACK for the first packet in the current window. When the ACK arrives, TCP's window opens up by at least one packet and thus a packet can be admitted in the send buffer. Once again the application can write a packet to the send buffer which is transmitted immediately without introducing any output latency.

Hence, as long as packets are not dropped in the network, MIN_BUF TCP will not add any output-buffering latency to the stream. Delay is added only as a result of packet dropping in the network. In this case, some packets that have already been admitted in the send buffer will have to be retransmitted, so these packets are delayed. Our experiments in Section 4.6 show that this delay is generally no more than the network round-trip time, which is much smaller than the standard TCP send-buffer delay.

MIN_BUF TCP removes sender-side buffering latency from the TCP stack so that low-latency applications are allowed to handle buffering themselves. This approach allows applications to adapt their bandwidth requirements to maintain low latency through data scalability techniques such as frame dropping, priority data dropping and dynamic rate shaping [94, 31, 64]. More precisely, the benefit of MIN_BUF TCP streaming is that the sending side application can wait longer before making its adaptation decisions (i.e., it has more control and flexibility over what data should be sent and when it should be sent). For instance, if MIN_BUF TCP doesn't allow the application to send data for a long time, the sending side can drop low-priority data. Then it can send higher-priority data which will arrive at the receiver with low delay (instead of committing the low-priority data to a large TCP send-buffer early and then losing control over quality adaptation and timing when that data is delayed in the send buffer). This approach trades data quality for timely delivery and is also called *adaptive quality of service* [94, 95, 65].

Note that in this work we do not modify TCP receive-side buffering because we expect applications to aggressively remove data from the receive-side buffer. Thus, receive-side delay is only as issue when packets are retransmitted by TCP. This issue is discussed further in Section 4.3.1.

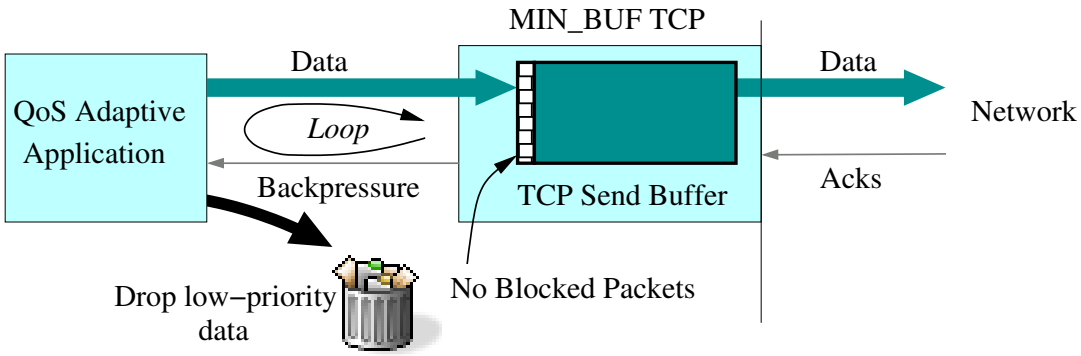
The next section analyzes how MIN_BUF TCP flows affect throughput and then extends our MIN_BUF approach to allow trading between latency and throughput. In addition to output latency, we are also interested in end-to-end latency for applications that stream data with MIN_BUF TCP. Section 4.3 describes two key factors, packet dropping and TCP congestion control that affect the end-to-end latency of MIN_BUF TCP flows. Then, Section 4.4 describes the implementation of MIN_BUF TCP flows in TSL. Section 4.5 explains our methodology for evaluating this technique and Section 4.6 presents our results. Finally, Section 4.8 justifies our claims about the benefits of our buffer-size tuning approach for low-latency streaming over TCP.

4.2 Effect on Throughput

The size of the send buffer in standard TCP is large because it helps TCP throughput. To understand this issue, consider that our send-buffer size adaptation approach will impact network throughput when standard TCP could have sent a packet but there are no new packets in the MIN_BUF TCP's send buffer. This condition can occur for several reasons. First, with each ACK arrival, standard TCP has a packet in the send buffer that it can send immediately. With MIN_BUF TCP, the send buffer size is limited to CWND, so it must inform the application and the application must write the next packet before it can send it as shown in Figure 4.3. With this loop, system timing and other components of input latency such as preemption latency and scheduling latency can affect MIN_BUF TCP throughput. Second, delayed and back-to-back ACK arrivals exacerbate this problem. Finally, the same problem occurs when TCP increases CWND.

These adverse affects on MIN_BUF TCP throughput can be reduced by adjusting the buffer size so that it is slightly larger than CWND. To clearly understand how much the buffer size should be increased, we need to consider events in TCP which cause new packets to be transmitted. There are four such events: ACK arrival, delayed ACK arrival, CWND increase and back-to-back ACK arrivals due to ACK compression [102]. These events and the way MIN_BUF TCP can handle them are described below.

ACK Arrival: Consider a stream of TCP packets where a contiguous set of CWND packets are currently being transmitted and constitute the current TCP window. When an ACK arrives for the first packet in the window, the window moves forward and admits a new packet in



With MIN_BUF TCP, arriving ACKs cause TCP to ask the application for the next packet because there are no "blocked" packets (see Figure 4.2). Thus an additional loop is introduced compared to standard TCP. This loop can reduce MIN_BUF TCP throughput.

Figure 4.3: System timing affects MIN_BUF TCP throughput.

the window, which is then transmitted by TCP. In this case, the MIN_BUF TCP send buffer should buffer one extra packet in addition to the CWND packets to avoid the loop shown in Figure 4.3. Then it can immediately send this packet upon ACK arrival instead of asking for the packet from the application.

Delayed ACK: To save bandwidth in the reverse direction, most TCP implementations delay ACKs and send, by default, one ACK for every two data packets received. Hence, each ACK arrival opens TCP's window by two packets. To handle this case, MIN_BUF TCP should buffer two additional packets.

CWND Increase: During steady state, when TCP is in its additive increase phase, TCP probes for additional bandwidth by increasing its window by one packet every round-trip time. Hence, TCP increments CWND by 1. At these times, the ACK arrival allows releasing two packets. With delayed ACKs, 3 packets can be released. Hence MIN_BUF TCP should buffer three additional packets to deal with this case and delayed ACKs.

ACK Compression: At any time CWND TCP packets are in transit in the network. Due to a phenomenon known as ACK compression [102] that can occur at routers, ACKs can arrive at the sender in a bursty manner. In the worst case, the ACKs for all the CWND packets

can arrive together.[1] To handle this case, MIN_BUF TCP should buffer 2 * CWND packets (CWND packets in addition to the first CWND packets). Note that the default send buffer size can often be much larger than 2 * CWND and thus we expect lower output latency even in this case.

If we take CWND increase into account with ACK compression, then TCP can send as many as CWND + 1 packets at once. Hence, in this case, MIN_BUF TCP should allow buffering 2 * CWND + 1 packets to achieve throughput comparable to TCP. In fact, we expect that if MIN_BUF TCP allows 2 * CWND + 1 packets in the send buffer, then its throughput should not differ from TCP throughput at all [74].

To study the impact of the send-buffer size on throughput and latency, we add two parameters A and B to MIN_BUF TCP flows. With these parameters, the send buffer is limited to $A*$CWND$+B$ packets at any given time. The send-buffer size is at least CWND because A must be an integer greater than zero and B is zero or larger (but less than CWND). Note that the parameters A and B represent a trade-off between latency and throughput. Larger values of A or B add latency but can improve throughput, as explained above. In general, we expect that for every additional CWND blocked packets, output latency will increase by a network round-trip time since a packet must wait for an additional CWND ACKs before being transmitted. These additional CWND ACKs, by the very definition of CWND, arrive over the course of a round-trip time.

From now on, we call a MIN_BUF TCP stream with parameters A and B, a MIN_BUF(A, B) stream. Hence, the original MIN_BUF TCP flow which limited the send-buffer size to CWND packets is a MIN_BUF($1, 0$) flow. For the send-buffer limit, we use two parameters instead of one because CWND is a variable and we wanted to test our hypothesis that a MIN_BUF TCP flow with 2 * CWND + 1 packets (or a MIN_BUF($2, 1$) flow) has the same throughput as a standard TCP flow.

Note that the buffer size of a MIN_BUF stream changes in an asymmetric manner when TCP changes the value of CWND as a result of its congestion control or avoidance algorithm. When CWND increases, the buffer size increases immediately. For example, if CWND increases by

[1] Once this occurs, TCP sends CWND+1 packets in a burst in response to all the ACK arrivals, which causes the ACKs to arrive in bursts again. Thus the burstiness in packet transmissions does not go away.

one, then a MIN_BUF$(1, 0)$ flow will allow the insertion of one additional packet in the send buffer immediately. However, when CWND decreases, the buffer size decreases slowly as packets are drained. For example, if CWND decreases by one, then a MIN_BUF$(1, 0)$ flow that had no blocked packets, will have one blocked packet since the MIN_BUF$(1, 0)$ buffer size has decreased by one. Once the blocked packet is transmitted, the new buffer size comes into effect.

4.3 Protocol Latency

In addition to reducing output latency, we are also interested in minimizing the end-to-end latency experienced by a network streaming application. In particular, we will be evaluating *protocol latency* for MIN_BUF TCP flows, which we define as the time difference from a write on the sender side to a read on the receiver side, both at the application level (i.e., socket to socket) latency. From this definition, it should be clear that protocol latency is composed of three components: output latency at the sender, network latency and input latency at the receiver. Our evaluation in Section 4.6 examines TCP protocol latency by measuring these three latencies at the sender side, on the network, and at the receiver side under various network conditions. Our results show that a substantial portion of protocol latency occurs on the sender side due to TCP's send buffer. Hence, MIN_BUF TCP flows can reduce protocol latency significantly.

Once MIN_BUF TCP flows remove output latency due to TCP's send buffer, latency due to packet dropping and TCP congestion control becomes visible in protocol latency. Their effects on protocol latency are described below.

4.3.1 Effect of Packet Dropping on Latency

When packets are dropped in the network, they have to be retransmitted by TCP. Due to the round-trip time needed for the dropped packet feedback, these retransmitted packets can add a round-trip delay for all packets in the send-buffer that follow them. Further, since TCP is an in-order protocol, the receiving side does not deliver packets that arrive out-of-order to the application until the missing packets have been received. Hence, a dropped packet adds at least an additional round-trip time to protocol latency for as many as CWND packets (i.e., protocol latency increases *by* a round-trip time *for* a round-trip time). We will see this effect in Section 4.6.4, when we

analyze the causes of protocol latency in detail.

4.3.2 Effect of TCP Congestion Control on Latency

TCP congestion control is the algorithm that TCP uses to adjust its CWND value and thus its transmission rate in response to congestion feedback. Normally, TCP perceives a network congestion event when it notices that a packet has been dropped. Hence, the effects of congestion control on protocol latency are similar to the effects of packet dropping on protocol latency. However, as we have seen above, packet dropping is guaranteed to introduce additional protocol latency and can lead to degraded throughput.

An alternative to using packet dropping as an implicit congestion event is an explicit congestion notification (ECN) mechanism for TCP [32]. With ECN, routers use active queue management and explicitly inform TCP of impending congestion by setting an ECN bit on packets that would otherwise have been dropped by the router. This ECN bit is received by the receiver and then returned in an ACK packet to the sender. The TCP sender considers the ECN bit as a congestion event and reduces CWND, and thus its sending rate, as shown in Figure 4.1, before packets are dropped in the network due to congestion. Hence, TCP enabled with ECN (TCP-ECN) can reduce network load and packet dropping in the network while allowing the continued use of TCP congestion control.

In essence, ECN allows TCP congestion control to operate even without packet dropping. Hence, let's consider how protocol latency is affected in MIN_BUF TCP flows when CWND changes independently of packet dropping, such as with TCP-ECN flows. When CWND increases during bandwidth probing, no output latency is added in the send buffer because a packet newly admitted into the send buffer can be sent immediately.

When CWND decreases, but not as a result of packet dropping, we need to consider two cases. First, when there are no blocked packets in the send buffer (i.e., the MIN_BUF$(1, 0)$ case), a reduction in CWND obviously does not affect output latency and protocol latency since there are no untransmitted packets that can get delayed in the buffer. A reduction in CWND causes the send-buffer to allow the application to write data later than it would have otherwise, but we assume that the application handles this delay by adapting their bandwidth requirements, as discussed earlier in Section 4.1. Note that, as explained above, with packet dropping, at least a *full* round-trip time

is added to protocol latency for the CWND packets in flight even if there are no blocked packets. Latency is added for packets that are retransmitted and for packets that arrive out-of-order on the receiver side.

Second, when there are blocked packets in the send buffer (i.e., the MIN_BUF(A, B) case where $B > 0$), a reduction in CWND introduces additional delay for these blocked packets. Assume that the value of CWND before it was reduced is $\mathrm{CWND_o}$ and the new reduced value of CWND is $\mathrm{CWND_n}$. Then the first blocked packet must wait for $\mathrm{CWND_o} - \mathrm{CWND_n}$ additional ACKs to arrive indicating that these many data packets have left the network before it can be transmitted [6, 73]. Assuming ACKs are paced (equidistant), they are RTT/$\mathrm{CWND_o}$ apart, where RTT is the round-trip time. Hence the delay added to the blocked packets is $(\mathrm{CWND_o} - \mathrm{CWND_n})/\mathrm{CWND_o} * \mathrm{RTT}$. Note that this delay is in addition to the delay that these packets would already have experienced if CWND had not changed. As an example, if $\mathrm{CWND_n}$ is half of $\mathrm{CWND_o}$, then *half* a round-trip time delay is added to the blocked packets. Note that when CWND is halved due to packet dropping, a full round-trip time delay is introduced in addition to this half round trip delay. Hence, in both cases (no blocked or blocked packets), CWND reduction via packet dropping adds an additional round-trip delay compared to CWND reduction without packet dropping.

A new algorithm called rate-halving [73] has been proposed that paces packet transmissions when CWND is changed. Instead of waiting for several ACKs before sending the first blocked packet, it paces the blocked packets at a rate slower than the arrival of ACKs (actually at half the rate since the new CWND is reduced to half the old CWND). Hence, the sending rate eventually converges from the old CWND to the new (and reduced) CWND rate. This approach avoids bursts and also helps to reduce the additional delay in the send buffer that we have calculated above. For example, when CWND is halved, the first blocked packet waits for two ACKs, the second for four ACKs, etc., instead of the first blocked packet (and thus all packets) waiting for CWND/2 ACKs in TCP and TCP-ECN. The experiments presented later in this chapter use this rate-halving technique, which has recently become available in the current versions of Linux.

4.4 Implementation

To implement adaptive tuning of the send-buffer size, we have made a small modification to the TCP stack on the sender side in the TSL kernel. This modification can be enabled per socket by using a new SO_TCP_MIN_BUF option, which limits the send buffer size to $A * \text{CWND} + B$ packets[2] at any given time, where A and B are parameters. By default A is one and B is zero for minimum send-buffer latency, but these values can be made larger with the SO_TCP_MIN_BUF option. With MIN_BUF TCP flows, the application is allowed to send when at least one packet can be admitted in the send buffer.

Correction for SACK TCP

Standard TCP uses cumulative acknowledgments in which received segments that are not at the left edge of the receive window are not acknowledged. This forces the sender to either wait a round-trip time to find out about each lost packet, or, if it is aggressive, to unnecessarily retransmit segments which have already been correctly received [30]. With the cumulative acknowledgment scheme, multiple dropped segments generally cause TCP to lose its ACK-based clock, reducing overall throughput catastrophically.

A Selective Acknowledgment (SACK) TCP helps to overcome these limitations. The receiving TCP sends back SACK packets to the sender informing the sender of data packets that have been received. The sender can then retransmit only the missing data segments. The number of selectively acknowledged packets that have been received by the sender in a round trip is kept in a variable called `sacked_out` in the Linux implementation of SACK TCP. When selective acknowledgments arrive, the packets in flight are no longer contiguous (i.e., do not have contiguous sequence numbers) but lie within a CWND+sacked_out packet window.

Our previous discussion regarding the send buffer limit applies for a non-SACK TCP implementation, where the packets in flight are in a contiguous window. For TCP SACK [72], we make a *sack correction* by adding the sacked_out term to $A * \text{CWND} + B$. We make the sack correction to ensure that the send buffer limit includes the non-contiguous window of packets in flight and is

[2] We assume that the size of each application packet is the maximum segment size (MSS).

thus at least CWND+sacked_out. Without this correction, TCP SACK is unable to send new packets for a MIN_BUF TCP flow and assumes that the flow is application limited. As a consequence, it reduces the congestion window multiple times after the arrival of selective acknowledgments.

4.5 Evaluation Methodology

In this section, we describe the tests we performed to evaluate the latency and throughput behavior of standard TCP and MIN_BUF TCP streams under various network conditions. All streams use TCP SACK and MIN_BUF TCP streams use the sack correction described in Section 4.4. We performed our experiments on a Linux 2.4 test-bed that simulates WAN conditions by introducing delay at an intermediate Linux router in the test-bed.

We experimented with three MIN_BUF TCP streams, MIN_BUF$(1, 0)$, MIN_BUF$(1, 3)$ and MIN_BUF$(2, 0)$[3], and compared their latency and throughput behavior with standard TCP. These streams should have increasing latency and throughput. A MIN_BUF$(1, 0)$ stream is the default stream which should have the least protocol latency. We chose a MIN_BUF$(1, 3)$ stream (which allows three packets in addition to the current packets in flight) to take ACK arrivals, delayed ACKs and CWND increase into account. Recall from Section 4.2 that these three events can cause a maximum of three packets to be released at the same time. Finally, we chose a MIN_BUF$(2, 0)$ stream because we expect it to have throughput close to TCP, as explained in Section 4.2. We expect that the average latency of a MIN_BUF$(2, 0)$ flow is about a round-trip time greater than a MIN_BUF$(1, 0)$ flow. However, it should have lower latency than standard TCP since TCP can buffer more than 2 * CWND packets.

We assume that low-latency applications use non-blocking read and write socket calls. These calls ensure, for example, that the sending side is not blocked from doing other work, such as media encoding, while the network is busy. In addition, the sending side can make adaptation decisions such as low-priority data dropping based on the failure of non-blocking write calls. With non-blocking calls, the protocol latency is measured from when the packet write is initiated on the sender side to when the same packet is completely read on the receiver side.

[3] In hindsight, we should have chosen a MIN_BUF$(2, 1)$ stream to account for the CWND increase also, as explained earlier.

The experiments in this chapter measure the improvement in protocol latency as a result of adaptive buffer tuning which reduces sender-side output latency. We ignore input latency on the sender-side kernel, output latency on the sender side and the processing latency incurred at the application level on the sender and receiver sides since this latter latency is application dependent. However, these latencies must also be included when studying the feasibility of a low latency application such as an interactive conferencing application. In Chapter 4.7, we perform additional experiments on a media streaming application to evaluate end-to-end latency at the application level.

4.5.1 Experimental Scenarios

Our first set of tests considers the latency behavior of TCP streams in a heavily loaded network environment. In such an environment the level of network congestion can change dynamically and rapidly with sudden bursts of incoming traffic. To emulate this environment, we run experiments with varying numbers of flows that trigger increase and decrease in congestion. For our experiments, we use three types of flows: 1) long-lived TCP flows, 2) bursts of short-lived TCP flows, and 3) a constant bit rate (CBR) flow, such as a UDP flow. The long-lived TCP flows are designed to simulate other streaming traffic. The bursts of short-lived TCP flows simulates web transfers. In our experiments, the small flows have fixed packet sizes and they are run back to back so that the number of active TCP connections is roughly constant [49]. The CBR flow simulates non-responsive UDP flows. While these traffic scenarios do not necessarily accurately model reality, they are intended to explore and benchmark the latency behavior of standard TCP and MIN_BUF TCP streams in a well characterized environment.

The second set of tests measures the relative throughput share of TCP and MIN_BUF TCP streams. Here we are mainly concerned with the bandwidth lost by MIN_BUF TCP traffic. These experiments are performed with the same types of competing flows described above. Third, we measured the CPU overhead of TCP and MIN_BUF TCP flows to understand the differences in the operational behavior between TCP and MIN_BUF TCP flows.

We are interested in several metrics of a latency-sensitive TCP flow: 1) protocol latency distribution, and specifically, the percentage of packets that arrive at the receiver within a *delay threshold*, 2) average packet latency, and 3) normalized throughput, or the ratio of the throughput

of a MIN_BUF TCP flow to a standard TCP flow. We choose two delay thresholds, 160 ms, which is related to interactive[4] streaming performance, and 500 ms, which is somewhat arbitrary, but chosen to represent the requirements of responsive media streaming control operations.

In addition to comparing the latency behavior of standard TCP and MIN_BUF TCP streams, we are also interested in understanding TCP-ECN's effect on protocol latency as described in Section 4.3.2. Our results describe how this "streaming friendly" mechanism affects protocol latency.

4.5.2 Network Setup

All our experiments use a single-bottleneck "dumbbell" topology and FIFO scheduling at the bottleneck. The network topology is shown in Figure 4.4. Each box is a separate Linux machine. The latency and throughput measurements are performed for a single stream originating at the sender *S* and terminating at the receiver *R1*. The sender generates cross traffic for both receivers *R1* and *R2*. The router runs `nistnet` [83], a network emulation program that allows the introduction of additional delay and bandwidth constraints in the network path. The protocol latency is measured by recording the application write time for each packet on the sender *S* and the application read time for each packet on the receiver *R1*. All the machines are synchronized to within one millisecond of each other using NTP.

We chose three round-trip delays for the experiments and conducted separate experiments for each delay. These delays were 25 ms, 50 ms and 100 ms and they approximate some commonly observed delays on the Internet. The cable modem from our home to work has 25 ms delay. West-coast to west-coast sites or East-coast to East-coast sites in the US observe 50 ms median delay and west-coast to east-coast sites in the US observe 100 ms median delay [47].

We ran our experiments over standard TCP and TCP enabled with ECN. For each round-trip delay, two router queue lengths are chosen so that bandwidth is limited to 12 Mbs and 30 Mbs. The TCP experiments use tail dropping. For ECN, we use derivative random drop (DRD) active queue management [37], which is supported in `Nistnet`. DRD is a RED variant that is implemented efficiently in software. The *drdmin*, *drdmax* and *drdcongest* parameters of DRD were chosen to be

[4]We assume that the end-to-end delay tolerance of interactive streaming lies between 200-300 ms, so the rest of the latency is for the end points.

1.0, 2.0 and 2.0 times the bandwidth-delay product, respectively. DRD marks 10 percent packets with the ECN bit when the queue length exceeds *drdmin,* progressively increasing the percentage until packets are dropped when the queue length exceeds *drdcongest*. Unlike RED, DRD does not average queue lengths.

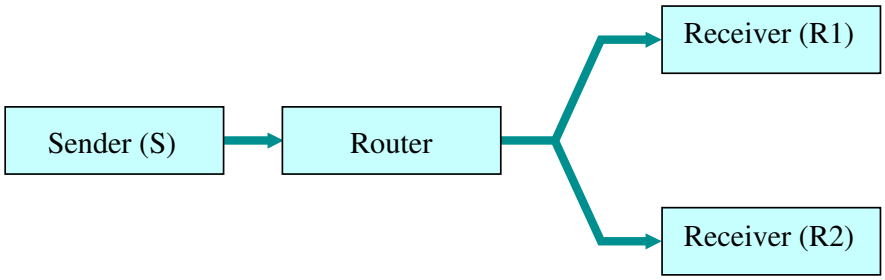


Figure 4.4: Network topology.

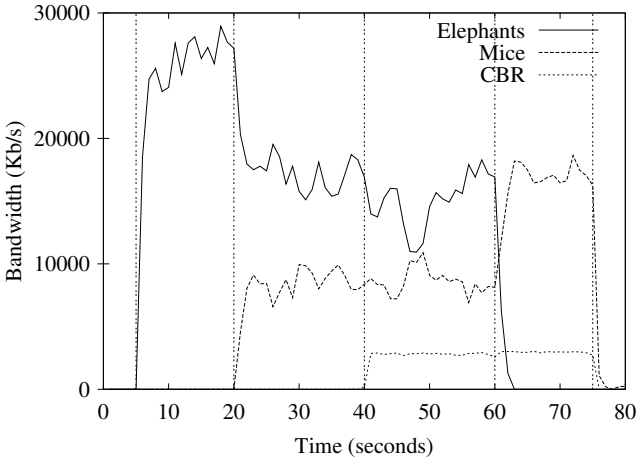
4.6 Evaluation

In this section, we discuss the results of our experiments. We start by showing the effects of using standard TCP and MIN_BUF TCP streams on protocol latency. Then we quantify the throughput loss of these streams. We investigate the latencies observed at the TCP sender, network and the TCP receiver and the causes of each latency. Finally, we explore using ECN enabled TCP to improve protocol latencies.

4.6.1 Protocol Latency

Our first experiment shows the protocol latency of TCP and MIN_BUF TCP streams in response to dynamically changing network load. The experiment is run for about 80 seconds with load being introduced at various different time points in the experiment. The standard TCP or MIN_BUF TCP long-lived stream being measured is started at $t = 0$ s. We refer to this flow as the *latency* flow. Then at $t = 5$ s, 15 other long-lived (*elephant*) flows are started, 7 going to receiver R1 and 8 going to receiver R2. At $t = 20$ s, each receiver initiates 40 simultaneous short-lived (*mouse*) TCP flows. A mouse flow is a repeating short-lived flow that starts the connection, transfers 20 KB of data, ends the connection and then repeats this process continuously [49]. The number of mouse flows

was chosen so that the mouse flows would get approximately 30 percent of the total bandwidth. At $t = 40$ s, CBR traffic that consumes 10 percent of the bandwidth is started. At $t = 60$ s, the elephants are stopped and then the mice and the CBR traffic are stopped at $t = 75$ s. Figure 4.5 shows the cross traffic (elephants, mice and CBR traffic) for a 30 Mbs bandwidth, 100 ms delay experiment. Other experiments have a similar bandwidth profile.

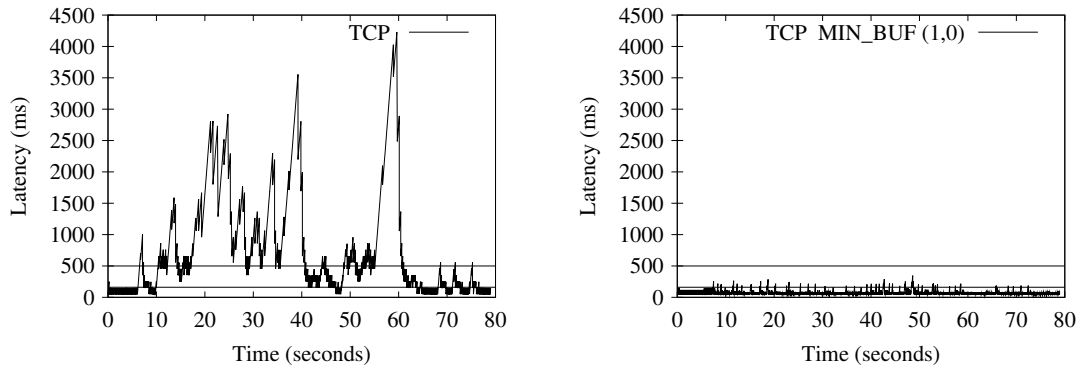


The cross traffic consists of 15 elephants that consume about 60-70% bandwidth when running together with mice, 80 mice consuming about 30% bandwidth and 10% CBR traffic.

Figure 4.5: The bandwidth profile of the cross traffic.

Figures 4.6 (a) and 4.6 (b) show the results of a run with a standard TCP and a MIN_BUF$(1, 0)$ stream when the bandwidth limit is 30 Mbs and the round-trip delay is 100 ms. Both these streams originate at sender S and terminate at receiver R1. These figures show the protocol latency of the latency flow as a function of packet receive time. The two horizontal lines on the *y* axis show the 160 ms and the 500 ms latency threshold.

Figure 4.7 shows the protocol latency of the three MIN_BUF TCP configurations. Note that in this figure, the maximum value of the y axis is 500 ms. These figures show that the MIN_BUF TCP streams have significantly lower protocol latency than a standard TCP stream. They show that, as expected, the MIN_BUF$(1, 0)$ flow has the lowest protocol latency while the MIN_BUF$(2, 0)$ has the highest protocol latency among the MIN_BUF TCP flows. The latency spikes seen in these flows are chiefly a result of packet dropping and retransmissions as discussed earlier in



(a) TCP

(b) MIN_BUF(1,0)

These figures show the protocol latency of packets plotted as a function of packet receive time. The bandwidth limit for this experiment is 30 Mbs and the round trip time is 100 ms. The horizontal lines on the figures show the 160 ms and 500 ms latency threshold.

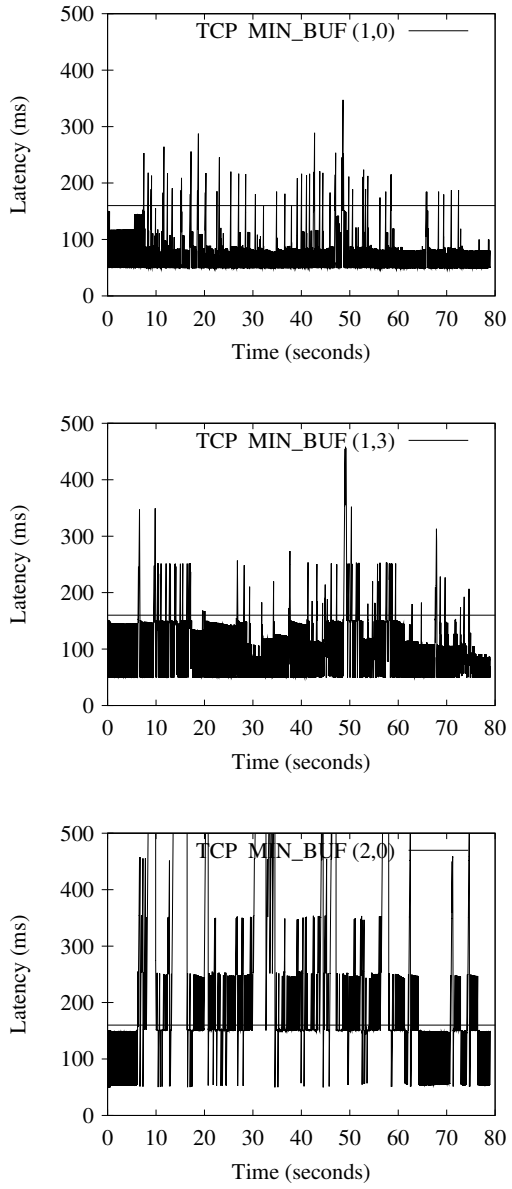
Figure 4.6: A comparison of protocol latencies of TCP and MIN_BUF(1,0) streams.

Section 4.3.1. We explore this issue in more detail in Section 4.6.4.

The protocol latency distribution for this experiment is shown in Figure 4.8. The experiment was performed with 30Mbs and 12Mbs bandwidth limits and with 100 ms, 50ms and 25 ms round-trip delays. Each experiment was performed 8 times and the results presented show the numbers accumulated over all the runs. The vertical lines show the 160 and 500 ms delay thresholds. The figures show that in all cases a much larger percent of TCP packets lie outside the delay thresholds as compared to MIN_BUF TCP flows. Note that the x-axis, which shows the protocol latency in milliseconds, is on a log scale. The figures show that, as expected, the percentage of packets with large delays increases with increasing round-trip delay and decreasing bandwidth. The underlying density of protocol latency for the 30 Mbs and 100 ms round-trip time experiment is shown in Figure 4.9. The density was calculated using 100 ms bins.

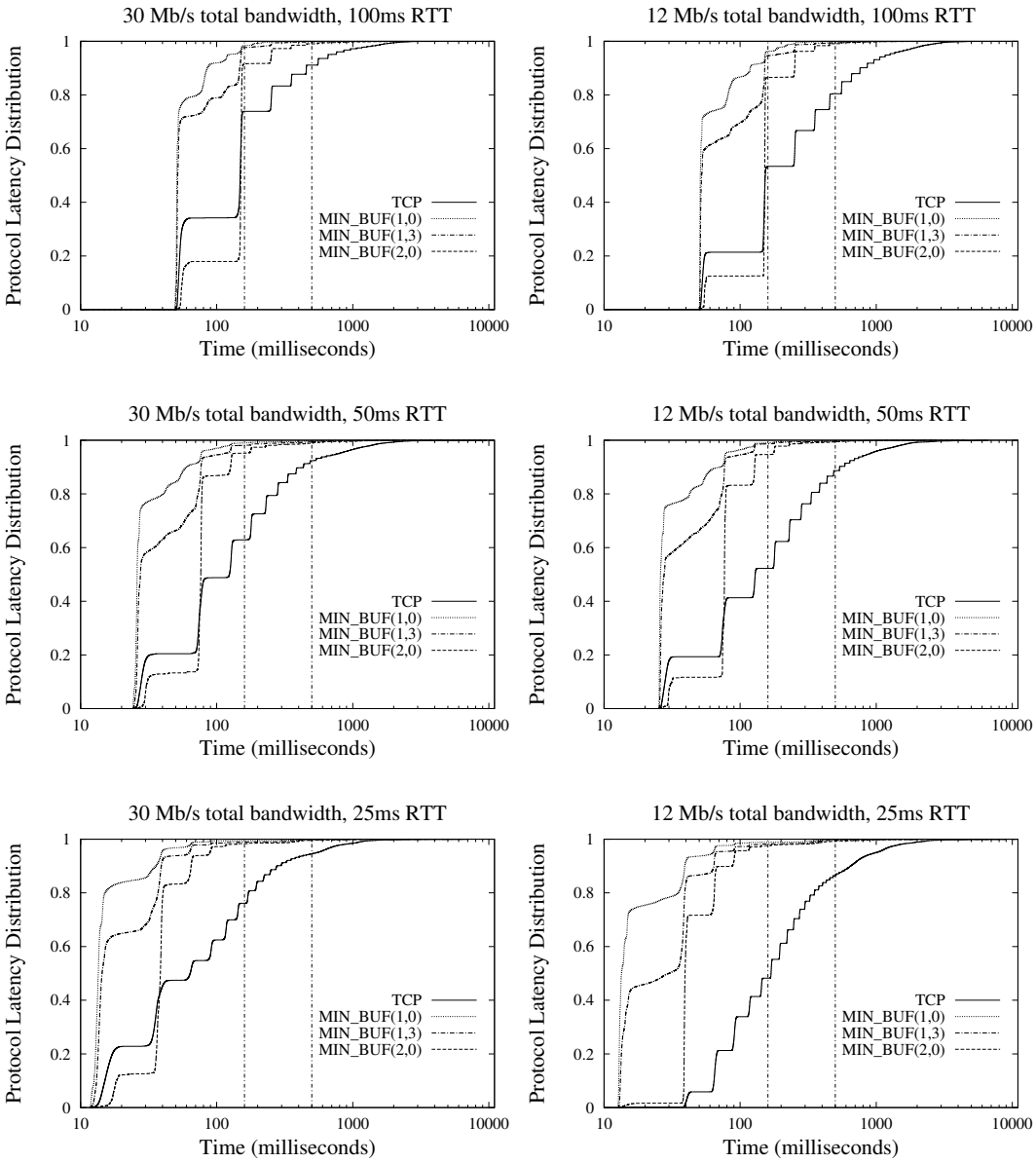
The percent of packets delivered within the 160 and 500 ms delay thresholds is summarized in Table 4.1. This table also shows that the packets delivered within the delay thresholds is very similar for the MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows.

The average (one way) protocol latency for each configuration is shown in Table 4.2. Each experiment was performed 8 times and these numbers are the mean of the 8 runs. The table shows



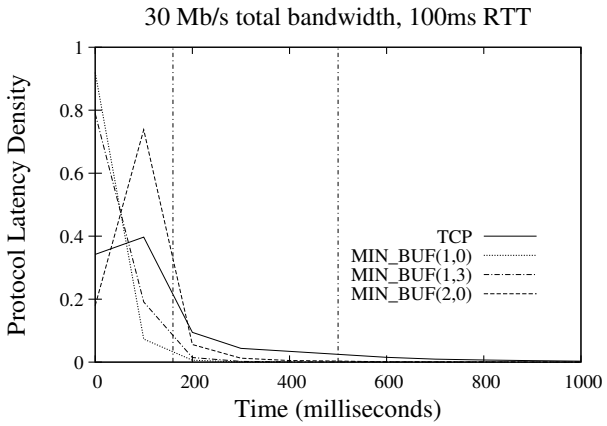
These experiments were performed under the same conditions as described in Figure 4.6. Note that the maximum value of the y axis is 500 ms, while it is 4500 ms in Figure 4.6. The figures from top to bottom show the latency of MIN_BUF(1,0), MIN_BUF(1,3) and MIN_BUF(2,0) flows respectively. For comparison, the maximum value of the y axis in the three graphs has the same value. The MIN_BUF$(2, 0)$ graph at the bottom has a few spikes above 500 ms, with the maximum spike being 3100 ms. As we will see later, MIN_BUF$(2, 0)$ flows are not well suited for low-latency streaming.

Figure 4.7: A comparison of protocol latencies of 3 MIN_BUF TCP configurations.



The experiment was performed with a 30 Mbs and 12 Mbs bandwidth limit and with 100 ms, 50ms and 25 ms round-trip delays. The vertical lines show the 160 and 500 ms delay thresholds. The x axis, which shows the protocol latency in milliseconds, is on a log scale. The figures in the left show the latency distribution when the bandwidth limit is 30 Mbs. The figures in the right show the latency distribution when the bandwidth limit is 12 Mbs.

Figure 4.8: Protocol latency distribution of TCP and three MIN_BUF TCP configurations.



The experiment was performed with a bandwidth limit and with 100 ms round-trip delays. The vertical lines show the 160 and 500 ms delay thresholds. The density is calculated as a histogram with a bin size of 100 ms.

Figure 4.9: Protocol latency density of TCP and three MIN_BUF TCP configurations.

		Delay = 100 ms		*Delay = 50 ms*		*Delay = 25 ms*	
Mbs	*Type*	*D160*	*D500*	*D160*	*D500*	*D160*	*D500*
30	std	0.77	0.92	0.61	0.91	0.78	0.92
30	m10	0.98	1.00	0.98	0.99	0.99	1.00
30	m13	0.98	1.00	0.97	0.99	0.98	1.00
30	m20	0.92	0.99	0.94	0.98	0.98	0.99
12	std	0.48	0.74	0.56	0.87	0.59	0.88
12	m10	0.93	0.99	0.97	0.99	0.98	0.99
12	m13	0.92	0.99	0.97	0.99	0.98	0.99
12	m20	0.84	0.98	0.93	0.99	0.97	0.99

The terms `std, m10, m13` and `m20` refer to standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ respectively. The terms D160 and D500 refer to a delay threshold of 160 and 500 ms.

Table 4.1: Percent of packets delivered within 160 and 500 ms thresholds for standard TCP and MIN_BUF flows.

that MIN_BUF TCP flows have much lower average latency and the deviation across runs is also much smaller. Note that the difference in the average latency between the MIN_BUF$(1,0)$ and the MIN_BUF$(2,0)$ flows is approximately the round-trip time. This difference is expected because MIN_BUF$(2,0)$ flows have CWND blocked packets which cause an additional latency of a whole

round-trip time as explained in Section 4.2.

Mbs	*Type*	*Delay = 100 ms*	*Delay = 50 ms*	*Delay = 25 ms*
30	std	226.31±0.87	218.84±40.34	138.61±21.0
30	m10	62.91±0.96	37.09±0.80	19.71±0.89
30	m13	76.19±2.71	51.54±3.73	28.29±1.70
30	m20	152.14±9.13	89.74±5.32	48.21±2.19
12	std	369.22±50.32	260.27±23.15	296.25±47.49
12	m10	69.73±2.15	38.50±1.09	25.94±1.80
12	m13	91.42±6.81	49.17±2.03	39.08±3.39
12	m20	162.26±6.06	87.90±1.46	61.31±5.59

The terms `std, m10, m13` and `m20` refer to standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ respectively. All average latency numbers (together with 95% confidence intervals) are shown in milliseconds.

Table 4.2: Average latency of standard TCP and MIN_BUF TCP flows.

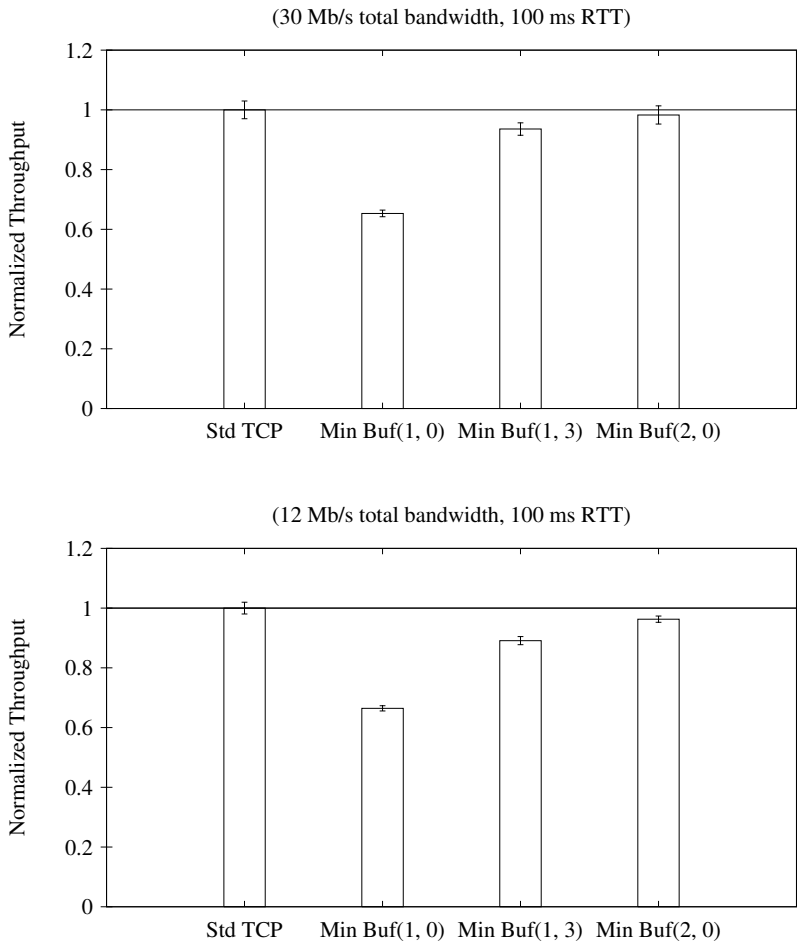
4.6.2 Throughput Loss

We are also interested in the throughput loss of MIN_BUF TCP streams. We measured the throughput of each of the flows as a ratio of the total number of bytes received to the duration of the experiment. Figure 4.10 and Table 4.3 shows the normalized throughput of the flows, which is the ratio of the throughput of the flow to the TCP flow. These numbers are the mean (and 95% confidence interval) over 8 runs. Here we have shown the normalized throughput numbers for the 30 Mbs and 12 Mbs experiments run with the 100 ms round-trip time. The numbers for the 50 ms and the 25 ms round-trip time experiments are similar and not shown here.

The figures show that, as expected, the MIN_BUF$(2,0)$ flows receive throughput close to standard TCP (within the confidence intervals). MIN_BUF$(2,0)$ flows have CWND blocked packets that can be sent after a packet transmission. So even if all current CWND packets in flight are acknowledged almost simultaneously, TCP can send its entire next window of CWND packets immediately. Thus we expect that MIN_BUF(2,0) flows should behave similarly to TCP flows.[5]

The MIN_BUF$(1,0)$ flows consistently receive the least throughput, about 65 percent of TCP.

[5]While we haven't done the experiments, we expect that the throughput of a MIN_BUF$(2,1)$ flow will be identical to a TCP flow as explained in Section 4.2.



The normalized throughput is the ratio of throughput of each flow to the ratio of a standard TCP flow.

Figure 4.10: The normalized throughput of a standard TCP flow and MIN_BUF TCP flows.

This result is not surprising because TCP has no new packets in the send buffer that can be sent after each ACK is received. TCP must ask the application to write the next packet to the send buffer before it can proceed with the next transmission. Thus, any input latency or application-processing latency will make the MIN_BUF$(1, 0)$ flow an application-limited flow as explained in Section 4.2. TCP assumes that such flows need less bandwidth and explicitly reduces CWND and thus the transmission rate of such flows.

The MIN_BUF$(1, 3)$ flows receive throughput 90-95 percent of TCP throughput. The three blocked packets in the send buffer handle delayed ACKs and CWND increase and thus reduce

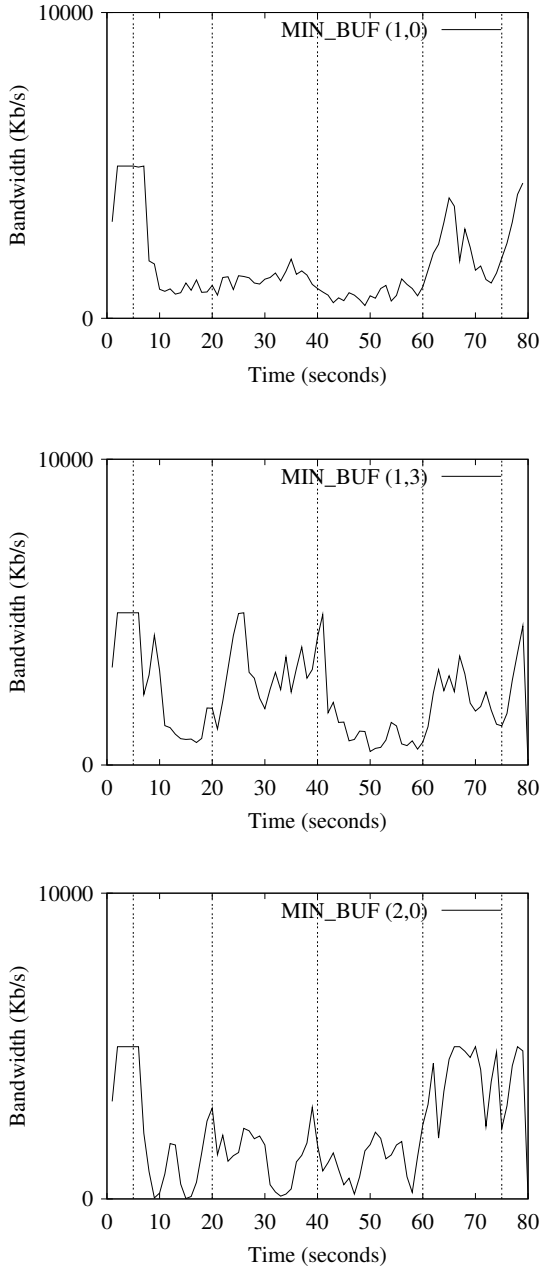
Mbs	*Type*	*Normalized Throughput*
30	std	1.000±0.03
30	m10	0.65±0.01
30	m13	0.94±0.02
30	m20	0.98±0.03
12	std	1.00±0.02
12	m10	0.66±0.01
12	m13	0.89±0.01
12	m20	0.96±0.01

Table 4.3: The normalized throughput of a standard TCP flow and MIN_BUF TCP flows when round-trip time is 100 ms.

the throughput loss due to the artificial application-flow limitation introduced by MIN_BUF$(1, 0)$ flows. We expect that the rest of the 5-10 percent throughput loss occurs as a result of ACK compression.

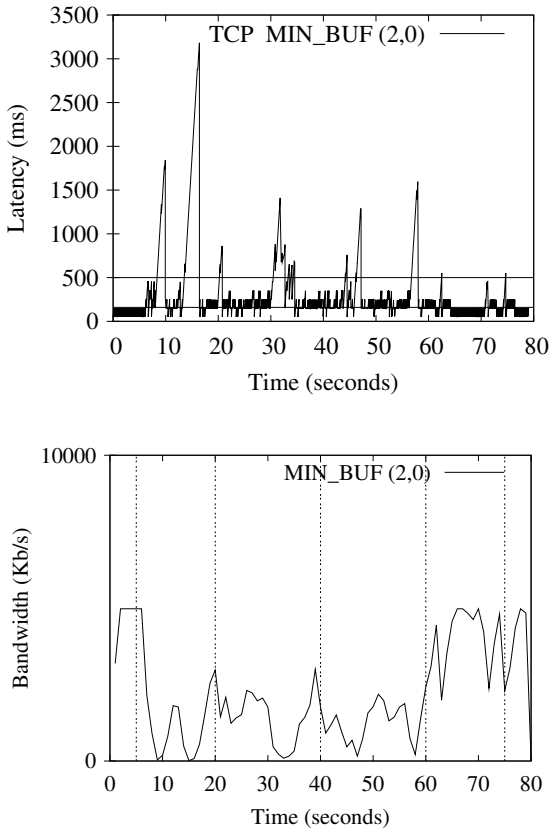
For a latency sensitive, rate-adaptive application, one metric for measuring the average flow quality would be the product of the percentage of packets that arrive within a delay threshold and the normalized throughput of the flow. This relative metric is related to the total number of packets that arrive within the delay threshold across different flows. Thus a larger value of this metric could imply better perceived quality. From the numbers presented above, MIN_BUF(1,3) flows have the highest value for this quality metric because both their delay threshold numbers (shown in Table 4.1) and normalized throughput numbers (shown in Figure 4.10) are close to the best numbers of the other flows.

Figure 4.11 shows the throughput profile of the MIN_BUF TCP flows for one experimental run. These figures provide several insights into the dynamic throughput and latency behavior of MIN_BUF TCP flows. First, the dips in the throughput of the MIN_BUF$(2, 0)$ flow are lower than the dips in the MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows. The MIN_BUF$(2, 0)$ flow is more aggressive because it is not prevented from expanding its window due to send-buffer underflow and thus immediately sends packets. However, it is also more bursty and periodically causes congestion and retransmission timeouts that temporarily produce large back-offs in sending rate. Second, the MIN_BUF$(1, 3)$ flow is able to probe for bandwidth much more effectively than the MIN_BUF$(1, 0)$ flow because the three blocked packets ensure that delayed ACKs and CWND



These figures from top to bottom show the bandwidth profile of MIN_BUF(1,0), MIN_BUF(1,3) and MIN_BUF(2,0) flows respectively for the experiment shown in Figure 4.7. The bandwidth profile is calculated as a histogram with a bin size of one second.

Figure 4.11: A comparison of bandwidth profile of 3 MIN_BUF TCP configurations.



For ease of comparison, these graphs are replicated from Figures 4.7 and 4.11. Note that the y-axis of the upper graph here is 3500 ms while it is 500 ms in Figure 4.7.

Figure 4.12: The protocol latency and bandwidth profile of a MIN_BUF(2,0) flow.

increase can be handled immediately and hence TCP transmissions are not limited by system or application timing. Note also that the protocol latency of the MIN_BUF$(2, 0)$ flow is highest when the flow throughput is lowest as shown in Figure 4.12 (the protocol latency and bandwidth graphs in Figures 4.7 and 4.11 have been copied here for ease of comparison). The reason is that the MIN_BUF$(2, 0)$ flow allows the send buffer to fill with CWND blocked packets and this buffer drains slowly when the bandwidth available to the latency stream goes down. This correlation is less obvious for MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows because they do not have as many blocked packets.

Figure 4.13 shows the standard TCP bandwidth profile. This profile is very similar to the one of the MIN_BUF$(2, 0)$ flow. It has several dips in throughput and these dips cause protocol latency

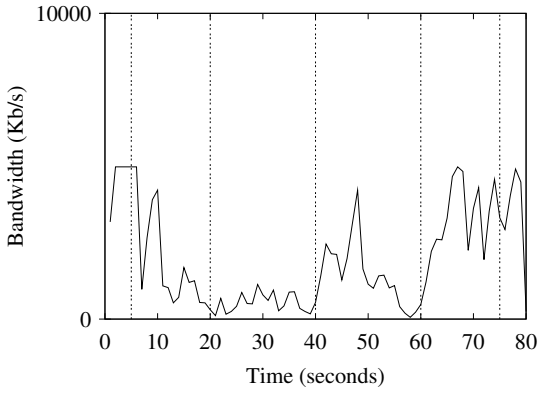


Figure 4.13: Bandwidth profile of a TCP flow.

	TCP	MIN_BUF(1,0)
write	2.33 seconds	2.67 seconds
poll	0.45 seconds	3.55 seconds

Table 4.4: Profile of major CPU costs in standard TCP and MIN_BUF TCP flows.

to shoot up as shown in Figure 4.6.

4.6.3 System Overhead

The MIN_BUF TCP approach reduces protocol latency compared to TCP flows by allowing applications to write data to the kernel at a fine granularity. However, this approach implies higher system overhead because more system calls are invoked to transfer the same amount of data. To understand the precise causes of this overhead, we profiled the CPU usage of TCP and MIN_BUF(1,0) flows for the experiment shown in Figures 4.6. The profiling showed that the two main costs on the sender side were the `write` and the `poll` system calls for both standard TCP and MIN_BUF TCP flows. Table 4.4 shows the total time spent in the kernel (system time) in `write` and `poll` for these flows.

These figures show that the MIN_BUF TCP flow has slightly more overhead for write calls. This result can be explained by the fact that MIN_BUF TCP writes one packet to the network at a time while standard TCP writes several packets at a time before the application is allowed to

write next time. In particular, TCP in Linux allows the application to write only after a third of the send buffer has been drained. Hence, in standard TCP, system overhead due to context switching between the application and the kernel is amortized over long periods of work.

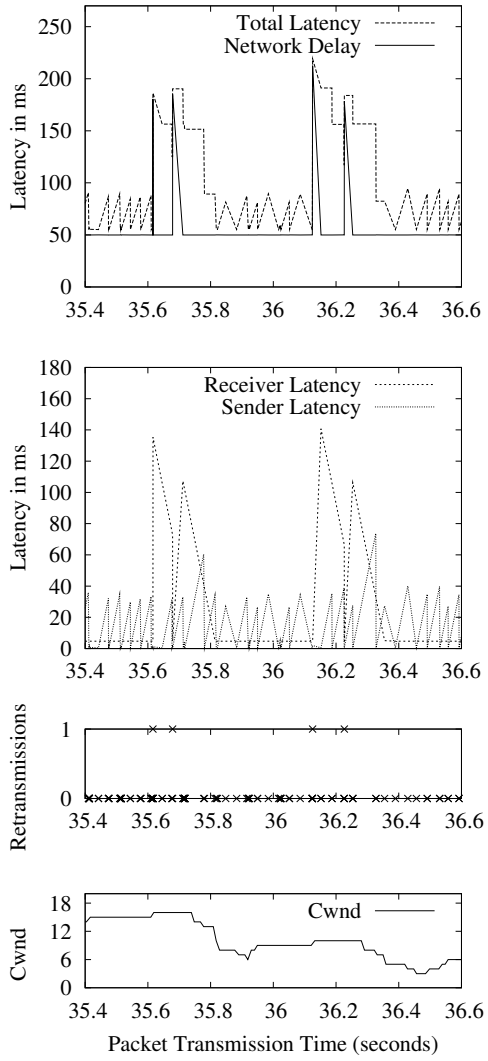
The second result shown in the figure is that MIN_BUF TCP has significantly more overhead for poll calls. The reason for this result is that poll is called for MIN_BUF TCP flows many more times than for standard TCP flows. With standard TCP, poll is called after the application writes a third of the send buffer (because, as explained above, TCP wakes the application only after draining a third of the buffer) while with MIN_BUF TCP, poll is called after every write (because MIN_BUF TCP wakes the application after each packet is sent to the network). The default send buffer size in TCP is 64 KB. A third of that size is 21 KB which allows 14 packets of MSS size (1448 bytes). Hence, in our application, with standard TCP, poll should be called after every 14 writes, while with MIN_BUF(1,0) it is called every time. We measured the number of calls to poll over the entire experiment for TCP and MIN_BUF(1,0) flows and found that the number was 6200 and 78500 respectively. The ratio of these numbers is 12.66, which is close to the expected value of 14. One reason for the slight discrepancy might be that when TCP increases CWND, then MIN_BUF TCP can send two packets in a single write, hence the ratio of writes will be slightly less than 14.

The numbers in Figure 4.4 above show that MIN_BUF TCP flows have a total of 2 to 3 times the system time overhead compared to standard TCP flows. This overhead occurs as a result of fine-grained writes that are allowed by MIN_BUF TCP flows. Note that these fine-grained writes are inherent with low-latency streaming but their benefit is that applications have much finer control over latency.

4.6.4 Understanding Worst Case Behavior

Figure 4.7 shows that MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows occasionally show protocol latency spikes even though they have small send buffers. To understand the cause of these spikes, we measured the delays experienced by each packet on the sender side, in the network and on the receiver side.

Figure 4.14 shows these delays for a small part of the experiment when packets were lost and retransmitted. The sender latency of each packet is the time from when an application writes to



This experiment was performed with a MIN_BUF$(1, 0)$ flow (30 Mbs bandwidth limit 100 ms RTT). All figures are plotted as a function of the packet transmission time. The third graph shows the occurrence of packet transmissions (crosses on 0 line) and retransmissions (crosses on 1 line). These figures show that the sender side latency is small for MIN_BUF$(1, 0)$ flows and that spikes in total latency occur primarily due to packet losses and retransmissions.

Figure 4.14: The packet delay on the sender side, the network and the receiver side.

the socket to TCP's first transmission of the packet. The network delay is the time from the first transmission of each packet to the first arrival at the receiver. The receiver latency is the time from the first arrival of each packet to an application read. Figure 4.14 shows that the latency spikes are

primarily caused by packet losses and retransmissions. In particular, the protocol (or total) latency does not depend significantly on the flow throughput or the congestion window size. For instance, the congestion window size at $t = 35.5$ ms and $t = 36.5$ ms is 15 and 4, but the total latency at these times is roughly the same.

Figure 4.14 shows that packet retransmissions initially cause the network delay to increase, followed by an increase in the receiver latency. The network delay increases by at least a round trip time, as explained in Section 4.3.1.[6] The receiver latency increases because TCP delivers packets in order and a lost packet temporarily blocks further packets from being released to the receiver application. The packets that experience receiver delay are exactly those that were sent after the dropped packet but before its retransmission. Note that the total latency remains high after a packet is dropped for approximately a round-trip time, as explained in Section 4.3.1. These findings motivated the need to explore mechanisms that can reduce packet dropping. One such mechanism that has been studied by the networking community is explicit congestion notification (ECN) [90, 97].

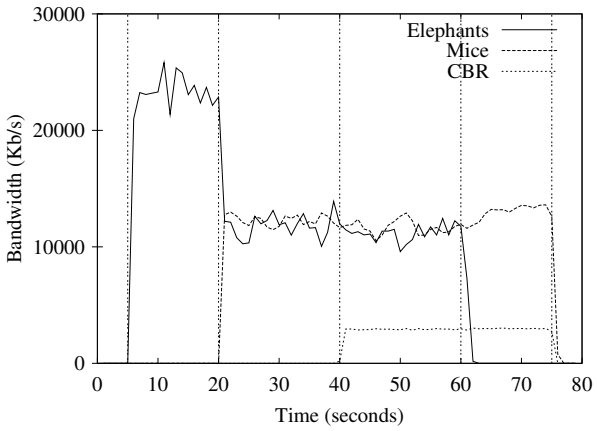
4.6.5 Protocol Latency with ECN

TCP with ECN is explicitly informed of impending congestion in the network, and it reduces its sending rate before packets are necessarily dropped in the network. We had explained in Section 4.3.2 that we expect protocol latency to improve with TCP-ECN. In particular, our analysis showed that for all MIN_BUF TCP flows, CWND reduction via packet dropping (TCP without ECN) adds an additional round-trip delay compared to CWND reduction without packet dropping (TCP with ECN).

In this section, we describe experiments that measure and compare the protocol latency of TCP-ECN flows and MIN_BUF TCP-ECN flows. We ran the same set of experiments as described in Section 4.6.1 but this time with ECN flows.

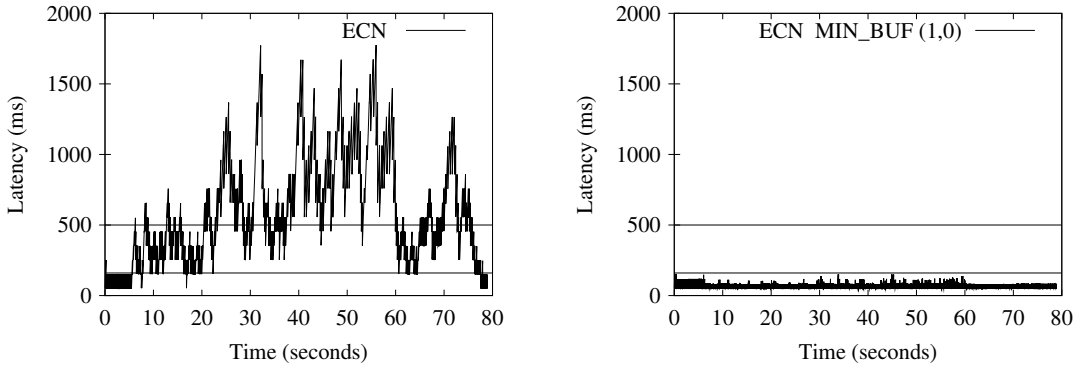
Figure 4.15 shows the bandwidth profile of the competing traffic. Figures 4.16 and 4.17 show the comparative protocol latencies. These figures are generated from experiments that are similar to those shown in Figure 4.6 except we enabled ECN at the end points and used DRD active queue

[6] Note that here we are accounting for the time spent in the send buffer due to retransmissions as part of network delay.



The cross traffic consists of 15 elephants that consume about 50% bandwidth when running together with mice, 80 mice consuming about 30% bandwidth and 10% CBR traffic.

Figure 4.15: The bandwidth profile of the cross traffic.



(a) TCP with ECN

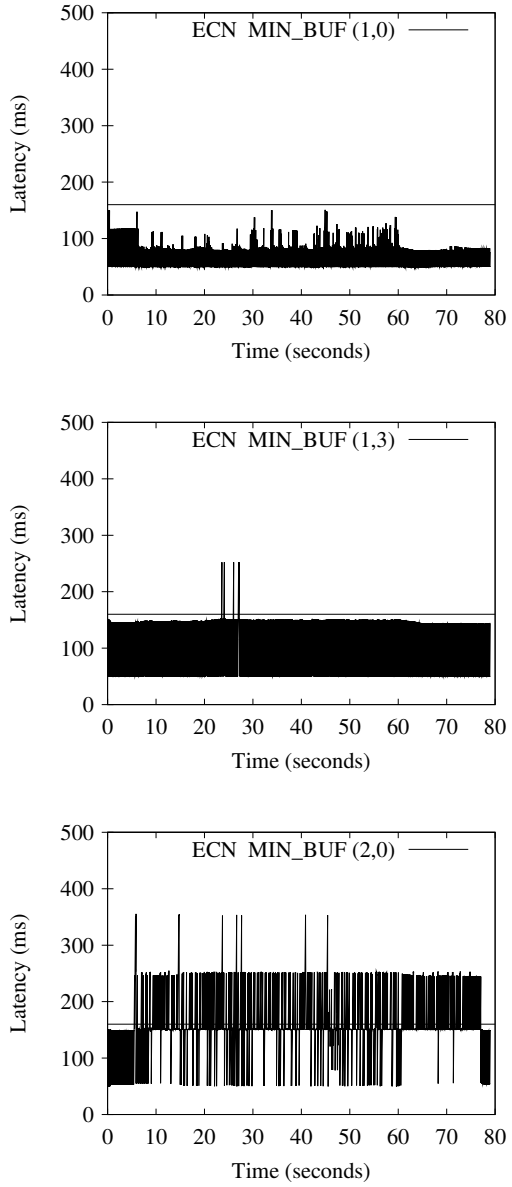
(b) MIN_BUF(1,0) with ECN

These figures show the protocol latency as a function of packet receive time. The bandwidth limit for this experiment is 30 Mbs and the round trip time is 100 ms. The horizontal lines on the figures show the 160 ms and 500 ms latency threshold.

Figure 4.16: A comparison of protocol latencies for TCP-ECN and MIN_BUF ECN streams.

management at the intermediate router.

These figures show that protocol latency is reduced in all MIN_BUF TCP-ECN cases as compared to standard TCP-ECN. In addition, MIN_BUF TCP-ECN has smaller and fewer latency



These experiments were performed under the same conditions as described in Figure 4.16. Note that the maximum value of the y axis is 500 ms, while it is 2000 ms in Figure 4.16. The figures from top to bottom shows the latency of ECN enabled MIN_BUF(1,0), MIN_BUF(1,3) and MIN_BUF(2,0) flows respectively.

Figure 4.17: A comparison of protocol latencies of 3 MIN_BUF TCP configurations with ECN.

spikes as compared to MIN_BUF TCP (shown in Figure 4.7). A close look at the raw data showed that ECN reduced packet dropping and retransmissions and thus had fewer spikes.

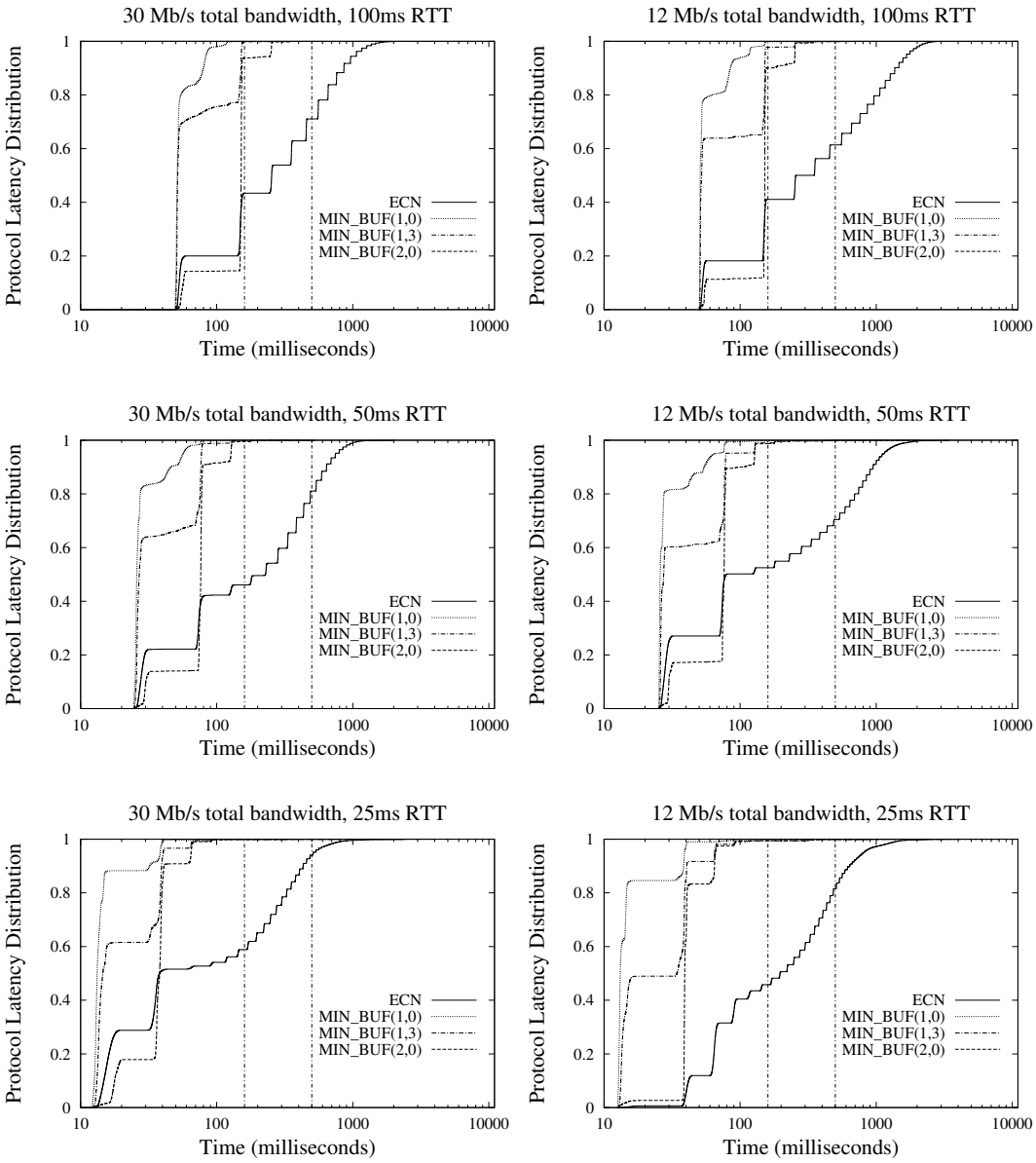
Figure 4.18 shows the protocol latency distribution for standard TCP-ECN and MIN_BUF TCP-ECN. When this figure is compared with Figure 4.8, it shows that these MIN_BUF TCP-ECN flows have smaller tails (i.e., they have a higher percentage of packets that arrive within the 160 ms and 500 ms thresholds).

The original motivation for active queue management and ECN was to avoid synchronized back-off effects in TCP. With drop-tail queuing in routers, bursts at a router would lead to packet dropping across a large number of TCP flows. All these flows would then reduce their transmission rate leading to poor network utilization. Eventually, they would all increase their rate, which would again lead to synchronized back-off. The use of ECN has been shown by several researchers to reduce synchronized back-off and to improve network utilization [97, 88]. Hence it is likely that ECN will be deployed in the future more extensively [90]. If deployed, it will interact favorably with MIN_BUF TCP.

4.7 Application-Level Evaluation

The previous section used micro-benchmarks to evaluate the output-buffering latency in the kernel due to TCP. This section evaluates the timing behavior of a real low-latency live streaming application and shows how MIN_BUF TCP helps in improving application-level end-to-end latency. In particular, our experimental results will show the end-to-end latency distribution of video frames, which helps determine the number of frames that arrived within a given deadline. In addition, we show that MIN_BUF TCP has lower variance in throughput compared to TCP, which allows streaming video with smoother quality.

We ran these experiments over TCP and MIN_BUF TCP on a best-effort network that does not guarantee bandwidth availability. With a non-adaptive media application, data will be delayed, possibly for long periods of time, when the available bandwidth is below the application's bandwidth requirements. Hence, we need to use an *adaptive* media application that can adapt its bandwidth requirements based on currently available bandwidth. For such an application, we chose the `qstream` application that has been developed in our group [64, 65, 46].



The experiment was performed with a 30 Mbs and 12 Mbs bandwidth limit and with 100 ms, 50ms and 25 ms round-trip delays. The vertical lines show the 160 and 500 ms delay thresholds. The x axis, which shows the protocol latency in milliseconds, is on a log scale. The figures in the left show the latency distribution when the bandwidth limit is 30 Mbs. The figures in the right show the latency distribution when the bandwidth limit is 12 Mbs.

Figure 4.18: Protocol latency distribution of TCP-ECN and three MIN_BUF ECN configurations.

We need to understand the *adaptive media format* and the *adaptation mechanism* used in `qstream` to analyze the components of application-level end-to-end latency. Qstream uses an adaptive media format called scalable MPEG (SPEG) that has also been developed in our group. SPEG is a variant of MPEG-1 that supports layered encoding of video data, which allows dynamic data dropping. In a layered encoded stream, data is conceptually divided into layers. A base layer can be decoded into a presentation at the lowest level of quality. Extension layers are stacked above the base layer where each layer corresponds to a higher level of quality in the decoded data. For correct decoding, an extension layer requires all the lower layers.

Qstream uses an adaptation mechanism called *priority-progress streaming* (PPS) [65]. For our purposes, the key idea in the PPS adaptation mechanism is an *adaptation period*, which determines how often the sender drops data. Within each adaptation period, the sender sends data packets in priority order, from the highest priority to the lowest priority. The priority label on a packet exposes the layered nature of the SPEG media so that higher layers can be sent and the display quality improved with increases in resource availability. Hence, the highest priority data has the base quality while lower priority data encodes higher quality layers. The data within an adaptation period is called an adaptation window. Data dropping is performed at the end of the adaptation period where all unsent data from the adaptation window is *dropped* and the server starts processing data for the next adaptation period. Consequently, the data that is transmitted within an adaptation period determines the quality of the presentation for that adaptation period. On the receiver side, data packets are collected in priority order for each adaptation period and then reordered in time order before they are displayed.[7] Note that if the available bandwidth is low or the sender side is blocked from sending data for long periods of time, then the sender can drop an entire adaptation window worth of data.

The minimum expected latency at the application level at the sender and the receiver sides is a function of the adaptation period. In particular, the sender must wait for an adaptation period to perform data prioritization before it can starting sending the adaptation window. Similarly, the receiver must wait for an adaptation period to receive and reorder an adaptation window. Figure 4.19 shows the application-level end-to-end latency experienced by the PPS streaming application. The

[7] PPS uses time-stamps on data packets to perform the reordering.

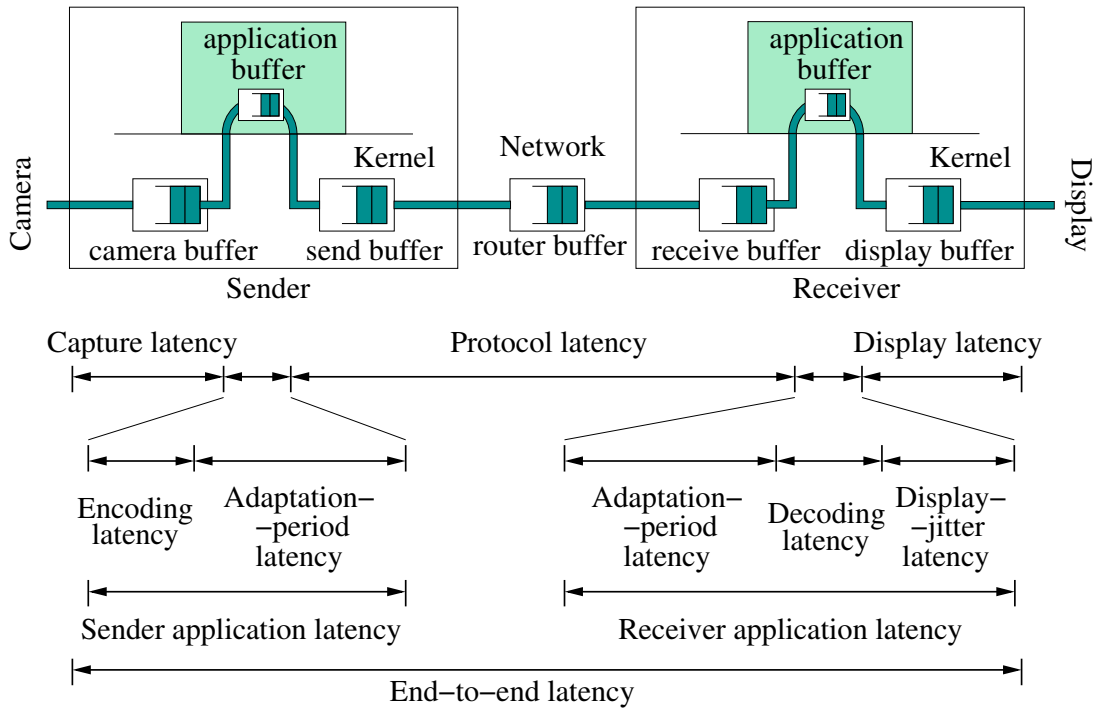


Figure 4.19: Breakup of end-to-end latency in the PPS streaming application.

end-to-end latency can be divided into capture latency, encoding latency, sender adaptation-period latency, protocol latency, receiver adaptation-period latency, decoding latency, display-jitter latency and display latency. We measure the end-to-end latency in our experiments. Note that capture, protocol and display latencies occur in the kernel while the rest of the latencies occur at the application level. In our experience, capture and display latencies are relatively small and hence our focus on protocol latency helps to significantly reduce kernel latencies. Our experiments do not directly affect the application latency at the sender or the receiver. However, note that decoding latency may become smaller when packets are dropped at the receiver.

4.7.1 Evaluation Methodology

We use three metrics *latency distribution, sender throughput* and *dropped windows* to evaluate the performance of the `qstream` application running under MIN_BUF TCP versus TCP. The latency distribution is the end-to-end latency experienced by all the adaptation windows during an

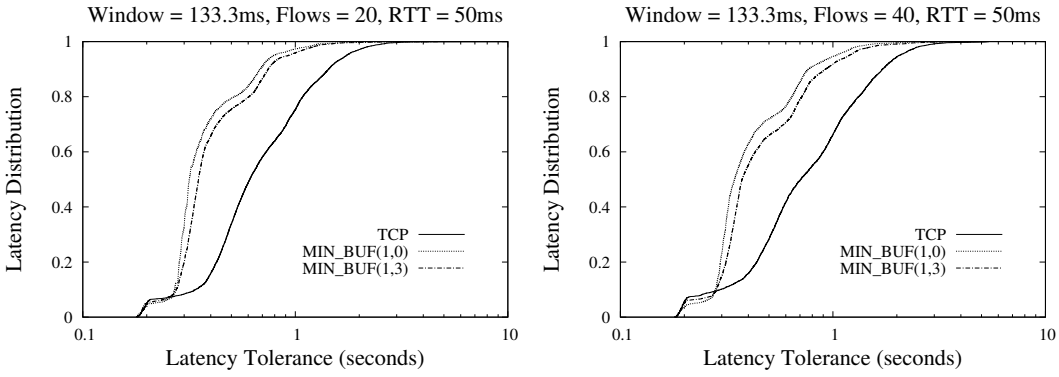
experimental run. To understand this notion, suppose that the end-to-end latency is fixed in the application, which is henceforth called the *latency tolerance*. The receiver drops data when an adaptation window arrives on the receiver side such that it cannot be displayed within the latency tolerance. The data that is dropped is the last part of an adaptation window and hence the lowest priority data in the window is dropped. The latency distribution measures the ratio of the number of windows that arrive within any given latency tolerance to the number of windows transmitted by the sender.[8] The sender throughput is the amount of data that is transmitted by the sender. Finally, dropped windows is the number of entire adaptation windows that are dropped by the sender. Note that dropped windows can be different even if the sender throughput is the same because the amount of data sent in each window (and hence the quality of a window) is variable. For the same value of sender throughput, a smaller value of dropped windows indicates that more windows were transmitted and the stream had smoother quality.

The experiments are performed on a Linux 2.4 test-bed that simulates WAN conditions by introducing a known delay at an intermediate Linux router in the test-bed. Experiments are run under varying network load with the cross-traffic being similar to the traffic described in Section 4.5.1. The experiments run on a single bottleneck "dumbbell" topology, which is also similar to the topology described in Section 4.5.2.

4.7.2 Results

First, we compare the latency distribution of flows using TCP versus MIN-BUF TCP. The figures for the experimental results shown below fix the round-trip time due to propagation delay to 50 ms and the bandwidth capacity of the link to 10 Mb/s. We performed experiments with other values of the round-trip time and the bandwidth capacity but those results are similar and hence not shown below. Figure 4.20 compares the results for TCP, MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows. It shows the latency distribution of each of the flows when the adaptation window period is 4 frames or 133.3 ms (camera captures data at 30 frames a second). The figures show that as long as the latency tolerance is less than a second, significantly more windows arrive in time for

[8] In `qstream`, the receiver displays data in a window that arrives within the latency tolerance and drops the rest of the data. The latency distribution metric only considers entire windows that arrive within the latency tolerance and hence is a conservative estimate of goodput, i.e., throughput that is useful.



(a) 20 competing flows

(b) 40 competing flows

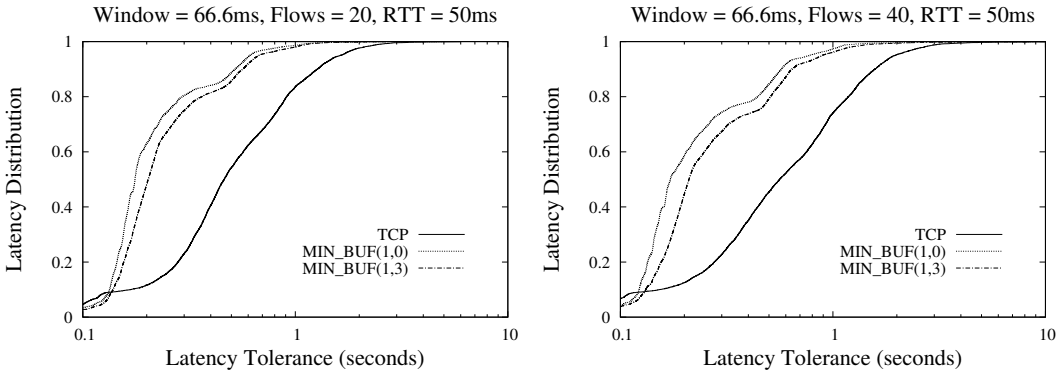
The bandwidth limit for all the experiments is 10 Mbs, the round trip time is 50 ms and the adaptation window is 133 ms. Note that for latency tolerance of less than a second, the MIN_BUF flows have significantly higher number of windows that arrive in time compared to TCP.

Figure 4.20: Latency distribution for different latency tolerances (adaptation window = 4 frames).

MIN_BUF flows compared to a TCP flow. For example, 80% windows arrive in less than 500 ms for MIN_BUF flows while only 40% windows arrive within the same time for a TCP flow when there are 20 competing flows.

We conducted experiments with different competing loads. Figures 4.20(a) and 4.20(b) show the results of experiments with 20 and 40 competing long-lived flows. These figures show that with increasing load, the percent of packets that arrive in time is only marginally affected for MIN_BUF flows (clearly the total amount of throughput achieved by the video flow is lower in the second case, as shown in Table 4.5 later).

Note that when the network is loaded, the end-to-end latency at the application level itself in `qstream` is at least 2 times the adaptation window or 266 ms. To reduce this latency, we ran the experiments above with a smaller adaptation window of 2 frames (or 66.6 ms). Figure 4.21 shows the latency distribution in this case. It shows that with MIN_BUF, 80% of packets arrive within 300-400 ms when there are 20 competing flows. This figure shows that the latency tolerance can be made tighter when the adaptation window is made smaller for any desired percent of timely window arrivals. The trade-off, as discussed below, is that the video quality varies more as the



(a) 20 competing flows

(b) 40 competing flows

The bandwidth limit for all the experiments is 10 Mbs, the round trip time is 50 ms and the adaptation window is 66 ms. Note that for latency tolerance of less than a second, the MIN_BUF flows have significantly higher number of windows that arrive in time compared to TCP.

Figure 4.21: Latency distribution for different latency tolerances (adaptation window = 2 frames).

	20 competing flows	40 competing flows
TCP	0.76 ± 0.11	0.56 ± 0.04
MIN_BUF(1,0)	0.60 ± 0.05	0.47 ± 0.04
MIN_BUF(1,3)	0.72 ± 0.04	0.54 ± 0.03

The bandwidth limit for all the experiments is 10 Mbs, the round trip time is 50 ms and the adaptation window is 66 ms. These results are the average of 5 runs.

Table 4.5: Throughput of a TCP flow and MIN_BUF flows with different competing flows.

window is made smaller.

Next, we compare the sender throughput of TCP and MIN_BUF flows. We show one set of results when the adaptation window is 2 frames since the size of the adaptation window does not affect the sender throughput much. Table 4.5 shows that the throughput achieved by MIN_BUF(1,0) is close to 80% of TCP while the throughput achieved by MIN_BUF(1,3) is close to 95% of TCP. These numbers are close to the relative throughput achieved by MIN_BUF flows in the micro-benchmarks presented in Table 4.10 and confirm that MIN_BUF(1,3) flows compete closely with TCP flows while significantly improving end-to-end latency.

	window = 4 frames	window = 2 frames
TCP	$45 \pm 3\%$	$58 \pm 5\%$
MIN_BUF(1,0)	$28 \pm 2\%$	$37 \pm 2\%$
MIN_BUF(1,3)	$27 \pm 2\%$	$34 \pm 3\%$

The bandwidth limit for all the experiments is 10 Mbs, the round trip time is 50 ms and the competing load has 20 long-lived flows. These results are the average of 5 runs.

Table 4.6: Throughput of a TCP flow and MIN_BUF flows with different competing flows.

The last metric we compare for TCP and MIN_BUF flows is the number of dropped windows which provides a rough estimate of the variation in video quality over time. Table 4.6 shows the ratio of adaptation windows dropped at the sender to the number of adaptation windows generated at the server, expressed as a percentage. The sender drops windows when it is unable to transmit data for close to an entire window period. This table shows that MIN_BUF flows drop fewer entire windows because these flows can send data in a less bursty manner, which results in smoother video quality since more frames are transmitted. Table 4.6 also shows that when the adaptation window period is small, then more windows are dropped as a percent of total number of windows because an idle transmission period is more likely to lead to a smaller window being dropped. Since larger numbers of windows are dropped with a smaller adaptation window, the video quality fluctuates more over time.

4.7.3 Discussion

The adaptation period allows control over the delay and the quality stability trade-off. A large adaptation period reduces the frequency of quality fluctuations because there are at most two quality changes in each adaptation period with priority-progress streaming and the window is more likely to be at least partially transmitted. However, a large adaptation period increases end-to-end latency as explained above. For a low-latency streaming application, the user should be provided control over the adaptation period so that the desired latency and quality trade-off can be achieved. However, note that this latency trade-off is application-specific and outside the scope of this thesis.

4.8 Conclusions

The dominance of the TCP protocol on the Internet and its success in maintaining Internet stability has led to several TCP-based stored media-streaming approaches. The success of TCP-based streaming led us to explore the limits to which TCP can be used for low-latency network streaming. Low latency streaming allows responsive streaming control operations, and sufficiently low latency streaming would make new kinds of interactive applications feasible.

This chapter shows that TCP's send buffer performs two functions that can be separated. In standard TCP, the send buffer keeps packets for retransmission and performs output buffering to match the application's rate with the network transmission rate. Output buffering helps to improve network throughput because, when ACKs arrive, packets can be transmitted from this buffer without requiring application intervention. However, output buffering adds significant latency.

For low-latency streaming, the send buffer should be used mainly for keeping packets that may need to be retransmitted since these packets do not add any additional buffering delay unless they are retransmitted. In addition, we showed that a few extra packets (blocked packets) help to recover much of the lost network throughput without increasing protocol latency significantly.

We showed that output-buffering latency is the largest source of latency in TCP. When this latency component is removed by using our MIN_BUF modification, latency spikes due to packet dropping and retransmissions become visible. TCP probes for bandwidth by expanding its transmission rate until it congests the network and packets are dropped. At this point TCP performs congestion control by reducing its rate. This behavior, which necessarily causes packet dropping also causes latency spikes. Hence, we explored the use of TCP with ECN, where routers inform TCP of congestion events. This approach allows using congestion control and bandwidth probing but does not require packet dropping. Our results show that MIN_BUF TCP with ECN has very low protocol latency and significantly improves the number of packets that can be delivered within 160 ms and 500 ms thresholds as compared to TCP or TCP-ECN.

Chapter 5

Real-Rate Scheduling

A CPU scheduling algorithm should ensure that low-latency applications are allocated CPU resources with predictable latency. Traditionally, real-time scheduling mechanisms such as *priority schemes* [67, 101] and *reservation-based schemes* [75] have been used to provide predictable and low scheduling latency. These schemes assume preemptive scheduling and their schedulability analysis assumes that preemption is immediate. Time-Sensitive Linux, which provides a responsive kernel with an accurate timing mechanism, allows implementing such CPU scheduling mechanisms more accurately and thus improves the accuracy of scheduling analysis.

A priority-based scheduling scheme executes the highest priority runnable thread at any given time. Two of the best known examples of priority-based scheduling schemes are rate-monotonic (RM) and earliest-deadline first (EDF) scheduling [67]. Priority schemes have a fundamentally difficult programming interface because different threads are most important at any given time, which requires dynamic adjustment of priorities of threads. To simplify this problem, RM assumes that all threads are periodic with fixed, statically known periods. It assigns fixed priorities to threads, with shorter period threads getting higher priority. To perform scheduling analysis, it assumes that the CPU requirements of each thread are known and the highest priority thread always voluntarily yields the CPU. In addition, the RM scheduling analysis is conservative which leads to under-utilization of the CPU.

To get around the under-utilization issue, EDF scheduling uses thread deadlines to dynamically adjust priorities of threads so that, at a given time, the thread with the earliest deadline is given the highest priority. In theory, this approach allows full CPU utilization. However, EDF, like RM, assumes that the CPU needs of each thread are known and the highest priority thread voluntarily yields the CPU. If the highest-priority thread does not voluntarily yield the CPU, both

RM and EDF priority schemes can cause starvation (i.e., they do not provide temporal protection to threads).

Reservation-based schemes such as proportion-period scheduling (PPS) avoid the problem of starvation. For example, PPS provides temporal protection to threads by allocating a fixed proportion of the CPU at each thread period. However, similar to priority-based schemes, it assumes that the CPU requirements of threads are known ahead of time. Hence threads must specify their proportion and their period to the scheduler before the scheduler accepts them.

In a general-purpose environment, there are several reasons why the resource requirements of threads may not be known statically. First, these requirements depend on the processor speed. For example, a real-time thread will generally require a smaller proportion of the CPU on a faster processor. Second, the resource needs are often data dependent. For example, video encoding and decoding times of variable-bit rate (VBR) streams such as MPEG depend on the size and the type of frames. Larger size frames take longer encoding and decoding times because more data is accessed. Also, certain types of frames in MPEG (I frames) can be decoded independently of other frames, while other types of frames (B and P frames), which are differentially encoded, depend of adjoining frames for correct decoding. In general, decoding times of independent frames is less than for dependent frames per byte of data because dependent frames must access more data. Finally, given that memory access times are significantly larger than cache access times, a thread's resource needs change dramatically based on the mix of applications running on the system, which affects cache pollution. With all these effects, resource specification becomes a complex problem. Hence, although proportion-period scheduling provides temporal protection and fine-grained control over resource allocation, it has not been widely accepted for general purpose OSs.

To use proportion-period scheduling in such an environment, a method of dynamically estimating the resource needs of low-latency applications is needed, so that their timing requirements can be met. The key insight here is that, from an application's point of view, there is a basic difference in its timing needs and its resource needs. Applications are aware of their timing needs. For example, a video application knows how often it needs to process data. However, as explained above, these applications do not know their resource needs. Consequently, a method for automatically determining a mapping between an application's timing needs and its resource needs is required. Such a mapping changes over time and hence has to be determined dynamically.

In this chapter, we present the design and implementation of a novel CPU scheduler, called *real-rate* scheduler, that uses feedback to determine an application's resource needs from its timing needs dynamically, and then automatically passes these resource needs to a proportion-period scheduler [104, 105, 40]. In our terminology, an application specifies its timing needs to the real-rate scheduler as a desired rate of *progress* by using application-specific *time-stamps*. These time-stamps indicate the progress rate desired by the application. For example, a video application that processes 30 frames per second can time-stamp each frame 33.3 ms apart. Ideally, the scheduler should ensure that every thread maintains its desired rate of progress towards completing its tasks. The key idea in our real-rate scheduler is to use the time-stamps to allocate resources so that the rate of progress of time-stamps matches real-time. Allocating more CPU than is needed will prevent other threads from using CPU time, whereas allocating less than is needed will delay this thread. In essence, the real-rate solution monitors the progress of threads and increases or decreases the allocation of CPU to those threads as needed.

To be effective, the real-rate approach needs the following: 1) a proportion-period scheduler that provides accurate and fine-grained control over proportion and period allocation, 2) an accurate estimator of an applications' resource requirements, and 3) an actuator that can adjust resource allocation frequently. As described in the beginning of this section, the first requirement is fulfilled by TSL. To satisfy the latter two requirements, resource monitoring, control and actuation must be done at a fine temporal granularity, which again requires a low-latency kernel such as TSL.

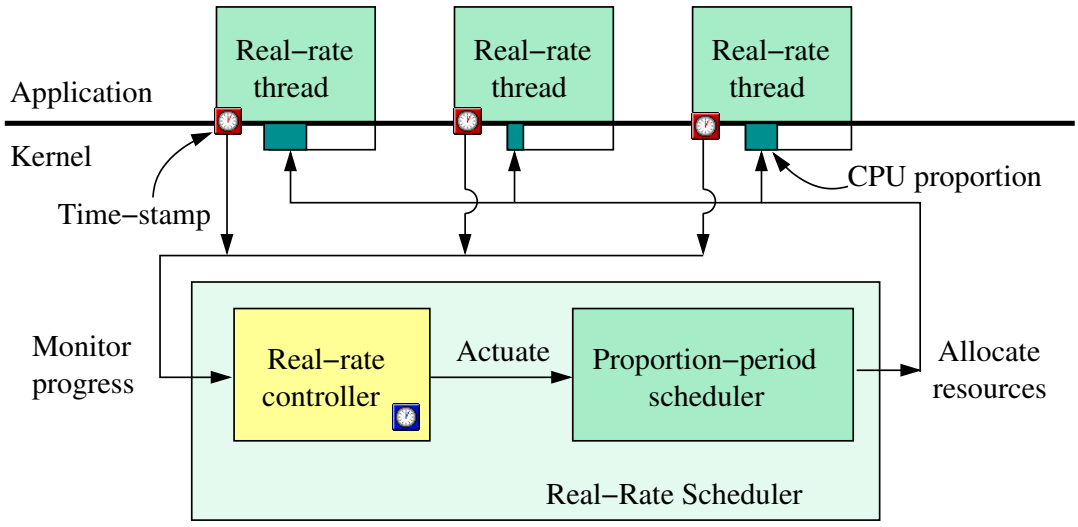
Note that standard reservation-based schemes such as proportion-period schedulers require threads to specify their requirements in resource specific terms, such as CPU cycles, which are hard to determine statically. In contrast, our approach requires applications to specify their timing requirements, which is easier for the application. The next section describes the design of the real-rate scheduler. Section 5.2 describes its implementation and Section 5.3 presents our evaluation. Finally, Section 5.4 concludes by justifying our claims about the benefits of our feedback scheduling approach.

5.1 Scheduling Model

Our feedback-based real-rate scheduling solution assigns proportions to threads dynamically and automatically as the resource requirements of threads change over time, based on monitoring the thread's progress. This monitoring is focused on the I/O or inter-process communication performed by threads since that is where the interaction with the external world takes place. To enable this approach, the scheduler requires I/O and IPC events to be *time-stamped*. These time-stamps capture the application's view of time: they should progress at the same rate as real-time (i.e., they indicate the progress rate desired by the application).

The basic feedback goal of the *real-rate* scheduler is to use these time-stamps to assign proportions to threads so that the rate of progress of time-stamps matches real-time. Given this goal, an accurate and responsive feedback controller will limit the instantaneous rate-mismatch between the application's notion of time and real-time. In addition, the accumulated mismatch over time between the two quantities is a measure of delay introduced in a real-rate thread and the controller can be tuned to minimize this value.

Figure 5.1 shows the high-level architecture of the real-rate scheduler. The scheduler consists of three main components: a proportion-period scheduler, a progress monitor and a real-rate controller. The proportion-period scheduler ensures that threads receive their assigned proportion of the CPU during their period. The progress monitor periodically monitors the progress made by these threads, which we call *real-rate threads*. These threads can be independent or their execution can be co-dependent, such as if they are connected by queues. Section 5.1.2 describes the monitoring function in more detail. Finally, the controller uses the thread's progress to adjust each thread's proportion automatically. We call this adjustment actuation or adaptation, since it involves tuning the system's behavior in the same sense that an automatic cruise control adjusts the speed of a car by adjusting its throttle. Note that the diagram resembles a classic closed-loop, or a feedback controlled system. The following subsections address each of the key components in the architecture.



The real-rate controller monitors the rate of progress of real-rate threads using time-stamps, and calculates new proportions based on the results. It actuates these values using a standard proportion-period scheduler.

Figure 5.1: Block diagram of the real-rate scheduler.

5.1.1 Proportion-Period Scheduler

The proportion-period scheduler in our architecture allocates CPU to threads based on two attributes: proportion and period. The proportion is specified in parts-per-thousand, of the duration of the period during which the application should get the CPU, and the period is the time interval, specified in microseconds, over which the allocation must be given. For example, if one thread has been given a proportion of 50 out of 1000 and a period of 10000 microseconds, it should be able to run up to 500 microseconds every 10000 microseconds.

A useful feature of proportion-period scheduling is that one can easily detect overload by summing the proportions: a sum greater than or equal to one indicates that the CPU is oversubscribed. If the scheduler is conservative, it can reserve some capacity by setting the overload threshold to less than 1. For example, one might wish to reserve capacity to cover the overhead of scheduling and interrupt handling.

The proportion-period scheduler can schedule both real-rate threads that have a visible metric of progress but do not have a known proportion and *reserved* threads that have a known proportion

and period. Reserved threads can directly specify these values to the proportion-period scheduler. Such a specification, if accepted by the scheduler, is essentially a reservation of resources for the thread. The scheduler does not allow the real-rate controller to modify these reservations. Instead, the controller is informed about the amount of CPU allocation that is available for real-rate threads. For example, if reserved threads use 40% of the CPU then 60% of the CPU is available (or less if the overload threshold is less than 1) to real-rate threads.

Upon reaching overload, the scheduler has several choices. First, it can automatically scale back the allocation of real-rate threads using some policy such as priorities, fair share or weighted fair share. Second, it can perform admission control on the reserved threads by rejecting or canceling threads so that the resulting load is less than 1. Third, it can raise exceptions to notify the threads of the overload condition so that they can adapt their CPU needs until these needs are no more than the overload threshold. These mechanisms are discussed in Section 5.1.3.5, where we explain the controller's policy for handling overload.

5.1.2 Monitoring Progress

The novelty of the real-rate approach lies in the estimation of application progress as a means of controlling CPU allocation. To estimate progress, the real-rate controller requires that each low-latency thread specify its timing requirements to the controller using time-stamps that indicates its desired rate of progress. For example, consider two threads with a producer/consumer relationship that use a shared queue to communicate with each other. Assume that the producer produces packets at a certain fixed rate and time-stamps these packets at the rate it is producing them. To avoid overflow the consumer must consume these packets at the same rate. The consumer can expose these packet time-stamps to the real-rate scheduler, which can then estimate the progress of the consumer by monitoring these time-stamps. If the consumer is consuming data slowly, the time-stamps of the packets entering the consumer from the queue will progress slowly and the scheduler will infer that the consumer needs more CPU. Similarly, when the time-stamps on the packets in the queue enter the consumer faster than real-time, the consumer can be slowed down. This analysis can also be extended to longer pipelines of threads where the scheduler infers each consumer's CPU requirements based on the incoming queue into the consumer.

Real-rate threads lie in user space while the real-rate controller is implemented in the kernel.

For the controller to monitor progress via time-stamps, we expose a memory area that is shared between each thread and the kernel. Threads specify their progress by writing time-stamps in this area while the controller in the kernel reads this area to determine application progress. This approach, which links application semantics to the kernel scheduler, is also called a symbiotic interface [104]. Figure 5.1 shows this interface as a clock that straddles the application and the kernel boundaries. Given a symbiotic interface, we can build a monitor that periodically samples the progress of the application, and feeds that information to the real-rate controller.

Our prototype scheduler provides two symbiotic interfaces. First, threads can explicitly create a specific shared memory mapping between the thread and the kernel using a new system call that we have implemented in TSL, which works similar to the `mmap()` system call. The application updates time-stamps in this memory area as it makes progress. This option can be used by threads that run independently of other threads and directly communicate with the external world such as a modem thread that receives and transmits modem data.

The second symbiotic interface is provided by a shared queue library. With this method, threads communicate with each other using the shared queue and insert time-stamped packets in the queue. The queue automatically creates the shared mapping described above and exposes these time-stamps to the kernel using the mapping. This method is used by threads that are connected in a pipeline. To use the shared queue interface, a thread in a low-latency application has three options. It can specify upon queue creation whether the kernel should monitor the time-stamp of the packet at 1) the head of the queue, or 2) the tail of the queue, or 3) neither (i.e., "don't monitor"). Normally, a thread that produces data into a queue (i.e., producer thread) specifies that the time-stamp on the tail or the last packet of the queue should be monitored. A thread that consumes data from a queue (i.e., consumer thread) specifies that the time-stamp on the head or the first packet of the queue should be monitored. The progress of the time-stamp on the tail packet indicates how quickly the producer is producing data into the queue, while the progress of the time-stamp on the head packet indicates how quickly the consumer is consuming data from the queue.

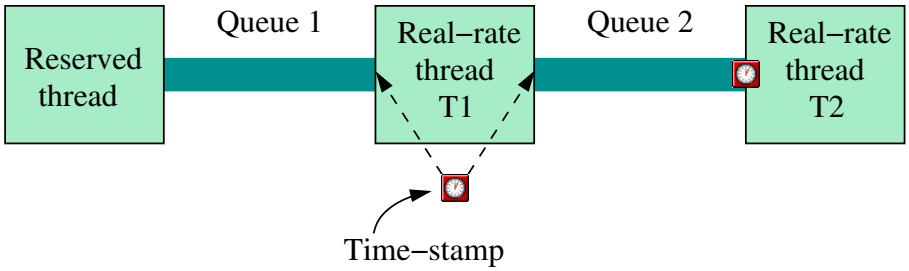
Threads specify the last "don't monitor" option to the queue for two reasons. First, they can be reserved (i.e., they require a fixed amount of CPU allocation, and hence the real-rate controller should not modify their proportion). Second, when threads are connected by multiple queues (say,

for example one or more input or one or more output queues or both), they may choose to specify one or more of these queues as progress indicators. The rest of the queues are specified with the "don't monitor" option so that the controller does not monitor these queues.

If a thread specifies that more than one of its queues should be monitored, then time-stamps from these queues must be aggregated to form a single "definitive" indicator of progress of the thread. Currently, the controller uses the minimum time-stamp among these queues as the definitive progress indicator. The rationale is that the CPU is the bottleneck resource, and its effect is visible in the slowest progressing queue. While this aggregation mechanism has sufficed for our applications, it would be easy to extend the queue interface so that different aggregation functions such as average or maximum time-stamp among queues can be used by the controller.

Figure 5.2 shows an example of a pipeline with three threads. In this example, the source thread is reserved and directly scheduled by the proportion-period scheduler. The thread in the center can choose to inform the kernel of its progress using either its incoming or its outgoing queue or both. The sink thread only has one queue and hence specifies the head of its incoming queue as its progress metric. Figure 5.3 shows pseudo code that would be used to implement the reserved and the real-rate threads. The `queue_open` function call registers each queue with the kernel so that the kernel can monitor time-stamps of packets in the queue. The `schedule_under` function chooses the scheduling policy. The function `set_time` sets the time-stamp on each packet while `send_packet` and `receive_packet` functions enqueue and dequeue packets from the queue.

In addition to our shared queue interface, it is possible to have other symbiotic interfaces that specify the timing behavior of applications to the kernel. For example, Unix-style pipes and sockets are essentially queues that are managed by the kernel. Real-rate threads using these kernel abstractions could use time-stamped packets and specify whether they are producers or consumers of these queues. In this case, similar to the API between applications and our user-level shared queue library, applications and the kernel would have to agree upon the packet format and the position where time-stamps occur in the packet. Then thread progress could be determined by simply monitoring sockets when data is written or read from them. This aspect of communication between user threads and the kernel is currently not implemented in TSL, but is part of our future



The pipeline consists of three threads, one reserved and two real-rate. The figure shows the three options available to real-rate threads. Producers of a queue specify that the time-stamp of the packet at the tail of the queue should be used for monitoring progress. Consumers of a queue specify that the time-stamp of the packet at the head of the queue should be used for monitoring progress. When a process has multiple queues, it can chose to inform the kernel of its progress using one or more queues. Also, threads that are reserved do not specify any progress monitor.

Figure 5.2: A real-rate pipeline with three threads.

work.

5.1.3 Real-Rate Controller

Real-rate threads have a visible metric of progress but do not have a known proportion. Hence, they do not specify their proportion to the proportion-period scheduler directly. Instead, the real-rate controller uses a monitoring component to measure the progress metric and assigns proportion to real-rate threads to ensure that they make their desired progress as specified by the time-stamps.

The following sections describe the design of the real-rate controller. The next section describes the real-rate proportion control mechanism. Section 5.1.3.2 describes the issue of time-stamp granularity that plays an important role in the choice of the sampling period of the controller. Section 5.1.3.3 explains how the period of real-rate threads is chosen in our system. Then, Section 5.1.3.4 motivates the choice of the sampling period of the controller. Section 5.1.3.5 describes the controller's behavior during resource overload, which occurs when the demands of real-rate threads exceed resource availability. Section 5.1.3.6 explains the method for tuning the parameters of the real-rate controller. Finally, Section 5.1.3.7 describes the controller composition behavior in our system (i.e., the interaction of real-rate control across different threads).

```
Reserved_thread(int proportion, int period)
{
  q1 = queue_open("queue 1", NO_MONITOR);
  schedule_under(PPS, proportion, period);
  while (1) {
     p = capture_packet();
     set_time(p, current_time);
     send_packet(q1, p);
     if (not late) sleep(period);
  }
}

Real_rate_thread1(int period)
{
  q1 = queue_open("queue 1", CONSUMER);
  q2 = queue_open("queue 2", NO_MONITOR);
  /* RRS is real-rate scheduler */
  /* period is time-stamp granularity */
  schedule_under(RRS, period);
  while (1) {
    p = receive_packet(q1);
    work_on_packet(p);
    send_packet(q2, p);
  }
}

Real_rate_thread2(int period)
{
  q2 = queue_open("queue 2", CONSUMER);
  schedule_under(RRS, period);
  while (1) {
    p = receive_packet(q2);
    work_on_packet(p);
  }
}
```

Figure 5.3: Code for the threads in the real-rate pipeline shown in Figure 5.2.

5.1.3.1 Proportion Control Mechanism

The *real rate* of a thread is defined as the interval between time-stamps monitored unit time apart. Based on this definition, when the real-rate is less than unity, the thread progresses slower than real-time and vice-versa. The goal of the real-rate mechanism is to maintain a *constant unit* real rate of progress. When this goal is met, the time-stamps of a thread will progress at the same rate as real-time, which reduces delays at each thread in the pipeline incurred due to rate mismatches.

To maintain the target unit real rate, the controller uses feedback to increase (or decrease) the proportion of the thread when the monitored real rate is less than (or greater than) unity.

The precise goal of the real-rate controller is to minimize the delay incurred at each thread. As we see below, this delay is related to the cumulative error in the real-rate of the thread and real time. The real-rate mechanism uses a discrete-time controller with a constant sampling period to maintain a target unit real rate. The choice of a constant sampling period controller is motivated by digital control theory, which allows using theoretical results to analyze the feedback mechanisms. Logically, each real-rate thread has a separate controller with its own sampling period. However, in practice, our controller is implemented as a single entity that performs the functions of each logical controller. The sampling period for each real-rate thread is chosen based on the granularity of time-stamps generated by the real-rate thread. The motivation for this choice is described in the next two sections where we also discuss the issue of granularity of time-stamps used to measure progress.

System Model To design a control law for real-rate threads, we first need to understand the model of the system or the system's dynamics. Figure 5.4 shows the system and control variables on a time-line. This figure shows sampling instants i and $i+1$ that are T_s time apart. At these sampling instants, the time-stamps that are monitored have values t_i and t_{i+1}. The proportion assigned to the thread between the two sampling instants is p_i. With these definitions, $t_{i+1} - t_i$ is the progress made by the thread T_s real-time interval apart. Hence, the real rate of the thread r_i between the two sampling instants is $(t_{i+1} - t_i)/T_s$.

The key assumption in our feedback-based real-rate model is that real-rate threads make progress that is approximately proportional to the CPU allocated to them. Hence, the relationship between the proportion of CPU allocated to them and their real-rate is defined as $p_i = g_i r_i$, where g_i is assumed to be a slowly changing variable that can be estimated. We will call the factor g_i the *proportional gain* of the thread. From the definitions above, the system model can be defined as follows: Since $p_i = g_i r_i$, therefore $r_i = p_i/g_i$. However, $r_i = (t_{i+1} - t_i)/T_s$. Rearranging the terms and replacing r_i in the two equations produces the system model shown in Equation 5.1 below.

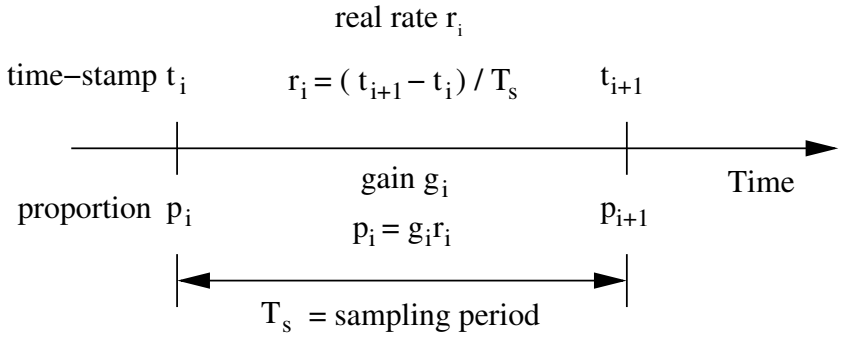


Figure 5.4: Time-line showing system and control variables.

$$t_{i+1} = t_i + \frac{T_s p_i}{g_i} \tag{5.1}$$

The system model equation defines the expected value of the thread time-stamp at sampling instant $i+1$, given the time-stamp at sampling instant i and the proportion p_i assigned at that time. This system model is a non-linear equation because the proportional gain g_i is not a constant, but an unknown variable whose value depends on time i. Next, this equation is used to derive the real-rate control law.

Control Law The *closed-loop* model of a control system is the dynamics of the combination of the system model and the control law that drives the system model. Ideally, we want to design a *linear* closed-loop model because a key advantage of such models is that their response and stability can be precisely determined using standard control theory. Hence a method for designing a control law for a non-linear system is needed that will yield a linear closed-loop model. One standard method for such design in feedback control is *feedback linearization*. In this approach, the control law for the controller is designed in such a way that it *inverts* the non-linearity in the system model. For a simple scalar non-linearity such as g_i, inversion is simply taking the reciprocal of the non-linearity, or $1/g_i$. Hence, the closed loop model that combines the controller and the system becomes linear and can be analyzed using standard control techniques. Appendix A describes the feedback linearization process in more detail. Here, we use this standard technique and apply Equation A.1 in Appendix A to the system model in Equation 5.1 to derive the control

law for the real-rate controller as shown in Equation 5.2.

$$p_i = (g_i/T_s)[t_{i+1}^{des} - t_i - (\alpha - 1)(t_i - t_i^{des})] \tag{5.2}$$

The proportion p_i is the proportion assigned to the thread at sampling instant i until sampling instant $i + 1$. The variable t_i is the time-stamp of the thread, monitored at sampling instant i and t_i^{des} is the desired value of the time-stamp. We describe how t_i^{des} is obtained based on the control goal below. The term $t_i - t_i^{des}$ is an error term because it is the difference between the actual output and the desired output of the system.[1]

Note that due to feedback linearization, the non-linear proportional gain term g_i is present in the control equation also. Below, we show how it can be estimated. Note also that substituting the control law for p_i in Equation 5.2 into Equation 5.1 for the system model yields the equation for closed-loop control dynamics. This closed-loop equation does not have the non-linear g_i term. Control theory analyzes system stability using the closed-loop equation and hence our real-rate control-system can be analyzed using linear-control techniques.

For stability, the gain parameter α in Equation 5.2 lies between 0 and 2 as explained in Appendix A.[2] The parameter α determines the responsiveness of this control equation. For example, larger values of α make the control equation more responsive because the proportion changes in larger steps for the same error term $t_i - t_i^{des}$. Section 5.1.3.6 discusses this issue further and describes how the control parameter α can be chosen.

To obtain t_i^{des} and t_{i+1}^{des}, we need to revisit the goal of the real-rate controller. The basic goal of the controller is to minimize the build up of delay at each real-rate thread. Assume that t_0 is zero. Then, delay will not build up at sampling instant i if the time-stamp value t_i is $i * T_s$ since the thread will have made real-rate progress equal to real-time. Hence, $t_i^{des} = i * T_s$. Note that even if t_0 is not zero, it would still be canceled out in the control Equation 5.2 since time-stamps are being subtracted from each other. Having derived the desired values of the time-stamps, control Equation 5.2 can be simplified as follows which finally yields the real-rate control Equation 5.3.

[1] Note that the time-stamp t_i is the output of the system but the input to the real-rate controller.

[2] Note that the parameter τ in the Appendix A is equal to $\alpha - 1$. Since $|\tau| < 1$, $0 < \alpha < 2$.

$$\begin{aligned} p_i &= (g_i/T_s)[t_{i+1}^{des} - t_i - (\alpha - 1)(t_i - t_i^{des})] \\ &= (g_i/T_s)[(i+1)T_s - t_i - (\alpha - 1)(t_i - iT_s)] \\ &= (g_i/T_s)[T_s + i\alpha T_s - \alpha t_i)] \\ &= g_i[1 + \alpha(i - \frac{t_i}{T_s})] \end{aligned} \tag{5.3}$$

This real-rate control equation shows that the proportion p_i is increased (or decreased) when the observed time-stamp t_i is less (or greater) than iT_s. When it is less, delay, which is equal to $iT_s - t_i$, is being accumulated and hence the allocation of the thread should be increased to speed up the thread.

System Parameter Estimation The control law in Equation 5.3 assumes that the proportional gain g_i is known. Recall from the previous section that we assumed that the relationship between the proportion of CPU allocated to a thread and its real-rate progress is linear, and defined as $p_i = g_i r_i$. Hence g_i is equal to p_i/r_i. Unfortunately, this value can only be known at time $i+1$ (and not at time i), since p_i has not yet been assigned at time i. In this sense, Equation 5.3 is non-causal because it depends on values in the future, and hence cannot be solved directly.

To get around non-causality, the real-rate control model assumes that the proportional gain g_i changes slowly over time, or $g_i \approx g_{i-1}$. Hence, g_i can be approximated as p_{i-1}/r_{i-1}, or $p_{i-1}T_s/(t_i - t_{i-1})$. Both these values are known at time i. In the absence of a precise model of the dynamic nature of g_i, we use a simple low-pass filter to estimate g_i based on past values of p and r. This approach reduces noise in the estimation at the expense of being less responsive to changes in g_i. In particular, $\hat{g_i}$ the estimate of g_i is derived as shown below.

$$\widehat{g_i} = (1 - \beta)\widehat{g_{i-1}} + \beta T_s p_{i-1}/(t_i - t_{i-1})$$

With this parameter estimation technique, the real-rate estimation and control laws can be expressed with Equations 5.4 and 5.5 as shown below. These laws together constitute the real-rate controller. Hence, from now on, the term real-rate controller will describe the combination of both these equations.

$$\widehat{g_i} = (1-\beta)\widehat{g_{i-1}} + \beta T_s p_{i-1}/(t_i - t_{i-1}) \quad (5.4)$$

$$p_i = \widehat{g_i}[1 + \alpha(i - \frac{t_i}{T_s})] \quad (5.5)$$

We choose the initial proportion gain $\widehat{g_0}$ at the start time to be 0. However, the proportion p_0 at the start time has to be a non-zero value to get the estimation started. Currently, we choose p_0 to be a small system-defined allocation value, 0.1% of total CPU. This choice can increase startup delay since the proportion allocated will be smaller than needed to make unit real-rate progress. However, our experiments show that this is not a serious problem because the proportion tends to increase rapidly (almost exponentially) in the beginning and hence the estimate converges quickly.

A second option is to start a thread with as much CPU proportion as is available at that time. Finally, it is possible to cache the proportion value of a thread from its past runs and use it to seed the initial proportion. These approaches will reduce startup delay.

The parameter β is chosen based on the expected noise in the sampling of the time-stamp t_i. We discuss this issue of choosing the α and β parameters in more detail in Section 5.1.3.6. Note that if the characteristics of g_i are known or can be modeled, then sophisticated techniques such as Kalman filters can be used for estimating g_i [61, 109].

There are two boundary conditions in the controller. First, if the time-stamp t_i does not increase and is equal to t_{i-1}, then the thread has made no progress in the last sampling period. In this case, since progress cannot be measured, $\widehat{g_i}$ is assigned the value of $\widehat{g_{i-1}}$ and p_i is assigned the value of p_{i-1} (i.e., the control state and output are not changed). The second boundary condition occurs when p_i becomes very small. Consider an example where p_i is assigned the value zero at sampling instant i. In that case, the thread will not be able to run and its progress cannot be measured correctly. Hence, the value of p_i should not approach 0. Currently, the minimum proportion a thread is assigned at any time is the same as the system-defined starting proportion value, or 0.1% of the CPU.

5.1.3.2 Granularity of Time-stamps

The granularity of time-stamps generated by a real-rate application has a significant impact on the performance of the real-rate controller. In particular, as explained in more detail below, the

controller cannot sample progress effectively at a sampling period finer than the timer granularity.

In general, the time-stamp granularity depends on application semantics. For example, a video encoding process may time-stamp every frame, in which case the time-stamp granularity is 33.3 ms for a 30 frames per second video. On the other hand, a CD quality audio encoder that generates 44100 samples per second may time-stamp every 100th sample (time-stamps on every sample would have high overhead) so the time-stamp granularity would be 2.27 ms.

To simplify the job of the controller and to tune the estimator and control law parameters, the real-rate controller expects real-rate threads to specify the approximate granularity of the time-stamps they will be generating. This thread parameter is specified to the controller using the symbiotic interfaces described in Section 5.1.2. In particular, the granularity is specified when the symbiotic interface is created or opened. In the future, we plan to determine time-stamp granularity automatically based on observing the progress of time-stamps.

5.1.3.3 Choosing the Period of Real-Rate Threads

The period of a real-rate thread specifies a deadline by which the scheduler must provide the allocation. It identifies 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 be impaired [23]. The period of the modem process can be chosen to be smaller than this value (for example, half this value) but a smaller period may introduce more overhead since dispatching happens more often.

Normally, real-rate threads in our system specify their period to the real-rate controller. When real-rate threads do not specify their period, we use the time-stamp granularity specified by the thread to determine an appropriate period for the real-rate threads. In particular, the thread period is made a multiple of the time-stamp granularity. More details about this choice are described in the next section where the choice of the sampling period is discussed. The precise value of the multiplicative factor depends on the controller response. Its value is determined using experiments described in Section 5.3.2.

5.1.3.4 Choice of Sampling Period

The sampling period of the controller determines its responsiveness. Finer-grained sampling allows the controller to be more responsive although it increases system overhead. However, the controller should sample no faster than the granularity of the time-stamps that indicate thread progress. Otherwise, the large time-stamp granularity will quantize the progress signal, which introduces error and potential instability in the feedback system.

This trade-off can also be described in terms of the estimator behavior in Equation 5.4. The estimator can accurately and more responsively track g_i when the parameter value β is large and the sampling period is small. However, time-stamp granularity introduces a disturbance in the measured value of g_i. To reduce this disturbance, β should be made smaller and the sampling period larger! This fundamental trade-off between accurate tracking and minimizing monitoring noise exists in *all* control systems [35]. We use experiments that are described in Section 5.3.2 to determine the optimal sampling period given the time-stamp granularity so that the real-rate controller can achieve its goal of minimizing delay. Based on these experiments, we set the sampling period to be a multiple of the time-stamp granularity specified by the application.

In addition to time-stamp granularity, the thread period also helps determine to the sampling period. Time-stamps in a real-rate thread increase when the thread does some minimum amount of logical work. The period of a real-rate thread is at least this amount of time. For example, a video decoding process may chose its period to be 33.3 ms because it must process a complete frame within that time for a 30 frames per second video. Hence, we expect time-stamps to increase every thread period. Based on this insight, we choose the controller sampling period for each real-rate thread based on two quantities, 1) a multiple of the time-stamp granularity, where the multiple is determined using experiments described later, and 2) a multiple of the thread period, where this multiple ensures that condition 1 is met. If the thread does not specify its period, then the thread period is set to the sampling period, which is determined using time-stamp granularity only.

More formally, if T_g denotes the time-stamp granularity, T_p the thread period, K is the multiplicative factor for T_g and L is the multiplicative factor for T_p, then there are two cases:

$$T_s = \begin{cases} LT_p \text{ where } (L-1)T_p \leq KT_g \leq LT_p & T_p \text{ is specified} \\ KT_g & T_p \text{ is unspecified} \end{cases} \quad (5.6)$$

Equation 5.6 shows that if T_p is specified, then the sampling period T_s is chosen to be a multiple of T_p so that it is just larger than KT_g. If T_p is not specified, then T_s (and T_p) are both made equal to KT_g. We define the term time-stamp *quantization* Q as shown below.

$$\begin{aligned} Q &= T_g/T_s \\ &= 1/K \end{aligned} \quad (5.7)$$

If time-stamps are very fine-grained or appear to be continuous, then they have a quantization of 0. If time-stamps are large as compared to the sampling period, then they have a large quantization. Our experiments will help determine the optimal quantization at which the controller minimizes thread delay. Since the time-stamp granularity T_g is specified by applications, Equations 5.7 and 5.6 can be used to determine the optimal sampling period.

An important reason why the sampling period and the process period are aligned is because sampling in between process periods would give a poor progress signal since the application would run at full CPU capacity for a while within its period and then stop based on its proportion. Hence the progress signal would appear to be a square-wave rather than a smooth signal.

5.1.3.5 Control Mechanism During Overload

Equation 5.2 updates a thread's proportion to maintain a target unit real rate. The resource needs of the real-rate threads in the system is the sum of the calculated proportions p_i across all threads. When this calculated resource utilization is greater than available resources or greater than the overload threshold OT, the resource is overloaded and all threads cannot be assigned their calculated proportions.

In the absence of an integrated quality adaptation mechanism, the controller handles resource overload by using priorities such that, in overload, the least important tasks are either suspended or dropped from the system entirely. This approach allows more important tasks to maintain

their real-rate progress.[3] If an integrated quality adaptation mechanism is available so that real-rate threads can adapt their resource needs when their progress needs are not met, then other approaches for handling overload such as *fair progress* [104] are possible. With fair progress, proportions are assigned to ensure that all threads achieve the same real rate. This approach is similar in spirit to a fair sharing policy where all threads are given equal resources. The difference is that instead of allocating equal proportions, fair progress ensures that all threads make the same progress, although at less than unit real-rate. Fair progress can be extended to weighted-fair progress in the same way fair sharing can be extended to weighted-fair sharing.

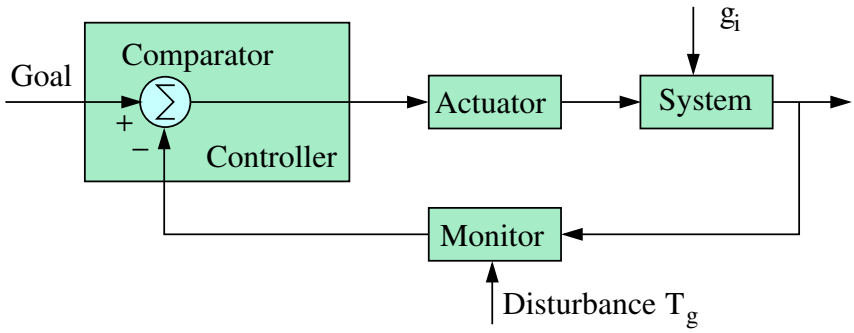
Our current TSL implementation does not pass information back to applications during overload. Several researchers have suggested using this upcall technique so that applications can adapt appropriately [85, 82]. While such techniques can be implemented in TSL, they require cooperation between applications and the OS and are outside the scope of this thesis.

When reserved threads, which are given a fixed proportion, are also executing in the system, their allocation must be subtracted from the overload threshold to obtain the amount of resources available to real-rate threads. In addition, the proportion-period scheduler performs admission control on reserved threads and does not admit new reserved threads if the total requirements of these threads would exceed the overload threshold.

5.1.3.6 Tuning the Real-Rate Controller

The goal of the real-rate controller is to minimize the delay introduced at each thread. This section describes qualitatively how the real-rate feedback controller is tuned to achieve this goal. Feedback control is the process of measuring a *control variable* and influencing the system so that the variable conforms to some *goal*. A block diagram of a control system is shown in Figure 5.5. In the real-rate control, the control variable is the real rate of the thread and the goal is unit real rate. The disturbance in the system is the variation in the proportional gain or g_i over time. The monitor measures time-stamps and feeds it to the controller, whose goal is to estimate this value accurately. The disturbance in the monitor is the time-stamp granularity T_g which makes the estimation of progress harder.

[3] If multiple tasks are assigned the same priority level, the scheduler prioritizes these tasks randomly.

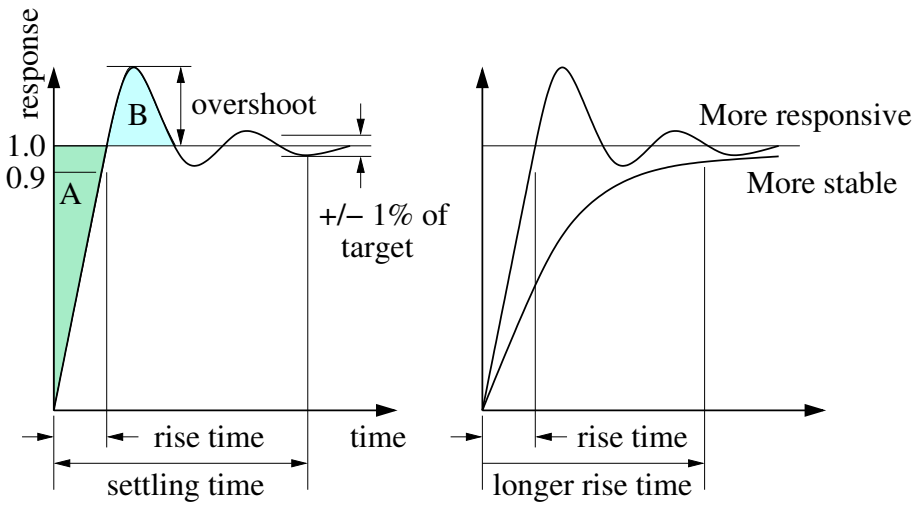


Feedback is used to counter the effects of disturbance in the system. The controller is system-independent and interfaces with the system via a system-dependent monitor and actuator. The actuator adjusts the system so that the monitored value of the control variable matches the goal.

Figure 5.5: A block diagram of a feedback control system.

Figure 5.6 shows the response of a control system to a *step input*. A step input is an external input that instantaneously changes the control variable from zero to one (normalized). It represents the worst case input for a feedback system. The response in Figure 5.6 shows how the system output changes over time. A system is considered more *responsive* than another system when it has a smaller *rise time*, which is the time it takes for the system to reach the vicinity of a new goal after the input has changed or been perturbed. A system is considered more *stable* than another system when it has a smaller *overshoot*, which is the maximum amount the output exceeds its final goal (as a percentage of the step size, which is one in Figure 5.6). Control tuning consists of systematically adjusting control parameters until the rise time and overshoot requirements can be met.

The real-rate mechanism in Equations 5.4 and 5.5 are analyzed using a step input as shown in Figure 5.6. The step input occurs due to an instantaneous increase in the processing needs of the thread. In other words, the value of the proportional gain g_i (the system disturbance) is increased in a step. If the proportion allotted to the thread is kept constant, the real rate $r_i = (t_i - t_{i-1})/T_s = (p_i/g_i)$ will decrease below unity since it takes longer to process data. Equation 5.5 will eventually increase the thread's proportion so that the real rate reaches its unit target. For the real-rate mechanism, the vertical axis in Figure 5.6 is the real-rate of the thread. It is normalized between zero and unity (i.e., immediately after the step input, the real rate is considered to be zero



The graph on the left shows the response of a control system to a step input (input changes from 0 to 1 at time 0). The response is characterized by the rise time, overshoot and the settling time. The graph on the right compares a responsive system (a small rise time) to an unresponsive system (no overshoot). Intuitively, the output responds to the input faster in a responsive system and thus the system is able to quickly track changes in the input. However, the responsive system has higher overshoot and exhibits more oscillatory behavior.

Figure 5.6: The control response.

and the response converges to unit real rate). To understand the effect of time-stamp granularity (the monitor disturbance), our experiments in Section 5.3.2 also vary time-stamp granularity T_g while inducing the proportional gain step.

The choice of α and β parameters in Equations 5.4 and 5.5 depends on the expected system disturbance (changes in g_i) and monitor disturbance (value of T_g). The α and β parameters affects the proportion overshoot and the additional delay experienced by the thread after a step input. Larger overshoot will cause faster resource overloading than needed, which will result in applications unnecessarily reducing their rate requirements. Thus one tuning goal is to reduce overshoot. The additional delay experienced by the thread is $iT_s - t_i$ in Equation 5.5. This delay is proportional to the shaded area A in Figure 5.6. The second tuning goal is to minimize this value or else applications may again unnecessarily reduce their rate requirements. Increasing α and β parameters decreases the additional delay but also increases the overshoot. In this thesis, we use simulation to tune α and β so that delay is minimized while at the same time overshoot is limited

to a reasonable value such as 15-30 percent used in standard control.

5.1.3.7 Feedback Composition

This section describes the interaction among the controls of different real-rate threads that are connected by shared queues and the buffering needs of these threads.

Controller Behavior The real-rate mechanism has been designed so that feedback interaction is minimized among threads in a pipeline. For example, the input to the real-rate control at a consumer thread is the time-stamp of the packet at the *head* of the queue (see the discussion in Section 5.1.2). A change in the incoming rate does not affect the head of the shared queue. Consequently, as long as the queue does not become empty, the real-rate controller behavior at the consumer thread is not affected by the controller behavior at the producer thread. The case when the shared queue becomes empty (and time-stamps do not increase) is treated as a boundary condition and the real-rate controller leaves the proportion unchanged so that when packets appear in the buffer next time, the thread's previous cached proportion is used. This policy reduces the ramp up time for allocating the correct proportion when packets become available but wastes resources when packets are not available. However, with properly tuned controls, we expect that buffers will not become empty and hence resources are not wasted.

Buffering Requirements We will consider the buffering needs at a queue shared between a producer thread T1 and a consumer thread T2 that are connected by the shared queue Q2 (see Figure 5.2). Note that the buffering needs are not in terms of bytes of data but in terms of time. For example, we say that the queue needs to store 100 ms of data. This method of viewing buffering requirements allows understanding the delay behavior in a pipeline since this queue can introduce 100 ms of delay.

The control response at the real-rate thread T1 affects the buffer requirements at queue Q2. Note that further downstream queues, if present, are not directly affected by the control behavior at thread T1 since the real-rate control response at thread T2 is not affected by the control response at thread T1, as described above. This property, which follows from our approach of minimizing interactions across the controllers of different threads, allows analyzing the buffering requirements

at a queue by considering the behavior of only its producer and its consumer.

To estimate the amount of additional buffer requirements at queue Q2 due to a step input at the thread T1, let's examine the left graph in Figure 5.6. During the rise time, the real rate of thread T1 is less than unity (although more data is being transmitted by thread T1 compared to before the step input!). Hence, the buffering needs (in time) at queue Q2 decreases by the shaded area A shown in Figure 5.6. However, during the time when the overshoot occurs, the real rate of the thread is greater than unity. The buffer length at queue Q2 increases by the shared area B shown in Figure 5.6. A similar analysis can be applied to thread T2 to show that the buffer requirements of queue Q2 increase during the rise time of thread T2. Hence, queue Q2 must provision for buffering based on the overshoot response of its producer thread T1 (shown by area A) and the rise-time response of its consumer thread T2 (shown by area B). With this amount of buffering, the real-rate scheduler will be sufficiently responsive that queues will not become empty or fill up, which can lead to additional delays. Note that this analysis assumes the worst case for delay because it assumes that T1 is in its overshoot phase when T2 is in its rise-time phase. If T1 and T2 are both in their rise-time or their overshoot phases, the buffering requirements will be smaller.

5.2 Implementation

To evaluate the real-rate CPU scheduling scheme, we have implemented a proportion-period CPU scheduler and the real-rate controller in Time-Sensitive Linux. The implementation of each is briefly described below.

5.2.1 Proportion-Period Scheduler

The proportion-period scheduler provides temporal protection to threads and uses an EDF scheduling mechanism to obtain full processor utilization. In a proportion-period scheduler with parameters (P, T) where P is the proportion and T is the period, the execution time Q is equal to $P*T$. In the implementation, a capacity and a deadline is associated with each proportion-period thread. At the beginning of each period, the capacity is recharged to the maximum value Q, and the deadline is assigned to be the end of this reservation period. In the EDF scheduler, all threads are sorted by this deadline and the first runnable thread in this sorted EDF queue is executed. In addition,

when two threads have the same deadline, the one with the smallest remaining capacity is scheduled to reduce the average finishing time. When a proportion-period thread executes, its capacity is decreased over time. Note that capacities can be decreased in the scheduling dispatcher that knows which thread is about to sleep and how long it has executed. When the capacity is zero, the reservation is depleted and the thread is blocked.

We use the high-resolution firm timers mechanism (see Chapter 2) in the proportion-period scheduler implementation for two purposes. First, a firm timer per-thread is used to detect the start of a reservation period. Second, we police a thread to detect when it has reached zero capacity by scheduling a system-wide firm timer in the scheduling dispatcher every time a reserved thread starts executing. When this firm timer fires, the reservation of the currently executing reserved thread is depleted. Our evaluation of the proportion-period scheduler in Section 5.3.1 shows that the firm timers mechanism detects the start of reservation periods and polices threads accurately with low latency.

The proportion-period threads executing under the EDF scheduler are dispatched by the standard Linux scheduler in TSL. All EDF scheduled threads generally have the highest real-time priority values in the system. The EDF threads themselves use an implicit two-level priority scheme where the next deadline thread has the highest priority and the other threads have lower priority. The key features of our proportion-period scheduler are very low overhead to change proportion and period, and fine-grain control over proportion and period values.

5.2.2 Real-Rate Controller

We have implemented the real-rate controller in the kernel as a Linux kernel module. The sampling period of the controller is equal to the process period. The proportion-period scheduler sets 100 μs as the limit of the sampling period. While this limit was chosen to ensure that the scheduler code does not itself take all the CPU and hence cause a livelock (since it doesn't run under the proportion-period scheduler), we have not yet needed to schedule applications at this granularity.

The controller samples the time-stamp of each thread at the boundary of the thread's period and then executes the control shown in Equation 5.2 to actuate a new proportion. If the time-stamp of a thread has not increased since the previous sampling period, the proportion is held constant.

When the real-rate requirement of a thread increases so that the total utilization across all

threads would cross the overload threshold, then the controller suspends the least important real-rate processes for overload control. By default, the overload threshold is set to 90% of CPU capacity.

The control equation is implemented using integer arithmetic because the Linux kernel does not save or restore floating point registers for kernel code to reduce the cost of context switches. Hence kernel code does not use floating point arithmetic. For high precision, proportions and periods in the kernel are maintained and manipulated in terms of processor cycles. This approach melds well with firm timers since firm timers maintain timer expiry values in processor cycles also. To minimize overhead, we use the minimum possible number of multiply and divide instructions to implement the controller. To do so, the α and β constants are chosen to be a reciprocal of powers of 2 (i.e., $1/2$, $1/4$, etc.).

Threads get scheduled under the real-rate controller when they use either of the two interfaces, described in Section 5.1.2, that allow the controller to monitor a thread's progress. In both cases, the controller shares a region with user code from where it obtains the thread's progress time-stamps.

5.3 Evaluation

The following sections evaluate the performance of our scheduler. Section 5.3.1 evaluates the overhead, correctness and accuracy of proportion-period scheduling behavior under TSL. Then, Section 5.3.2 evaluates the behavior of the real-rate controller.

5.3.1 Proportion-Period Scheduler

At the lowest level, the overhead of the proportion-period scheduler depends on the execution time of the following four actions: 1) scheduling dispatch, 2) waking up of a process, 3) processing at period boundaries, and 4) policing of allocations. Since the proportion-period scheduler uses an EDF mechanism, the execution time of these actions depends on three basic operations: 1) the time needed to get the earliest thread, 2) the time to manipulate threads in the sorted EDF queue and 3) the time needed to process firm timers. The scheduling dispatcher deletes the previous thread from the EDF queue and runs the next thread with earliest deadline. The waking up of

a process requires an insertion into the EDF queue. The processing at a period boundary and the policing of allocations requires firing a firm timer after which either a process is woken up or a scheduling dispatch operation is invoked. Section 2.4.2 showed that the overhead of firm timers is $O(log(n))$, where n is the number of active timers in a 10 ms interval and, in practice, it is very low. The cost of accessing the earliest deadline thread from the EDF queue is $O(1)$. The cost of inserting and deleting from an EDF queue is $O(log(n))$ where n is the number of active, runnable proportion-period threads in the system. The number of active runnable threads depends on whether the periods of different processes are aligned. Interestingly, when the periods of different processes are not aligned, the number n is small, because as long as the system is not overloaded, the average number of runnable threads in a sufficiently long time interval must be less than or equal to 1. When the periods of threads are aligned, several processes may be runnable at once, in which case the number of active runnable proportion-period threads can be as large as n.

Our original motivation for implementing a TSL system was to implement a proportion-period scheduler that would provide an accurate reservation mechanism for the feedback-based real-rate controller. The accuracy of allocating resources using a feedback controller depends (among other factors) on the accuracy of actuating proportions. In our initial proportion-period scheduler implementation on standard Linux, there were three sources of inaccuracy: 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, we ran two processes with proportions of 40% and 20% and periods of 8192 μs and 512 μs respectively on a 1.5 GHz Pentium-4 Intel processor with 512 MB of memory.[4] These processes were run

[4]The proportion-period scheduler implementation in these experiments is an older implementation that allowed thread periods to only be multiples of 512 us. While this period alignment restriction is not needed for a proportion-period scheduler, we were initially using it because it simplified the feedback-based adjustment of thread proportions during overload.

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 3.4). 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 [92].

Table 5.1 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 25 μs 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. Note that the numbers presented are for the maximum deviation in allocation (not the average) over all periods during the entire experiment. 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 [23] and thus could be supported on TSL at the application level even under heavy 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 us	0.3% ($\simeq$25 μs)	5 μs	6% ($\simeq$490 μs)	534 μs
Thread 2 Proportion: 20%, 102.4 μs Period: 512 us	0.7% ($\simeq$3 μs)	10 μs	4% ($\simeq$20 μs)	97 μs

Table 5.1: Deviation in proportion and period for two processes running on the proportion-period scheduler on TSL

Note that while the maximum amount of time stolen is 490 μs and 20 μs for threads 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. One way to improve the performance of proportion-period scheduling in the presence of heavy file system load is to defer certain parts

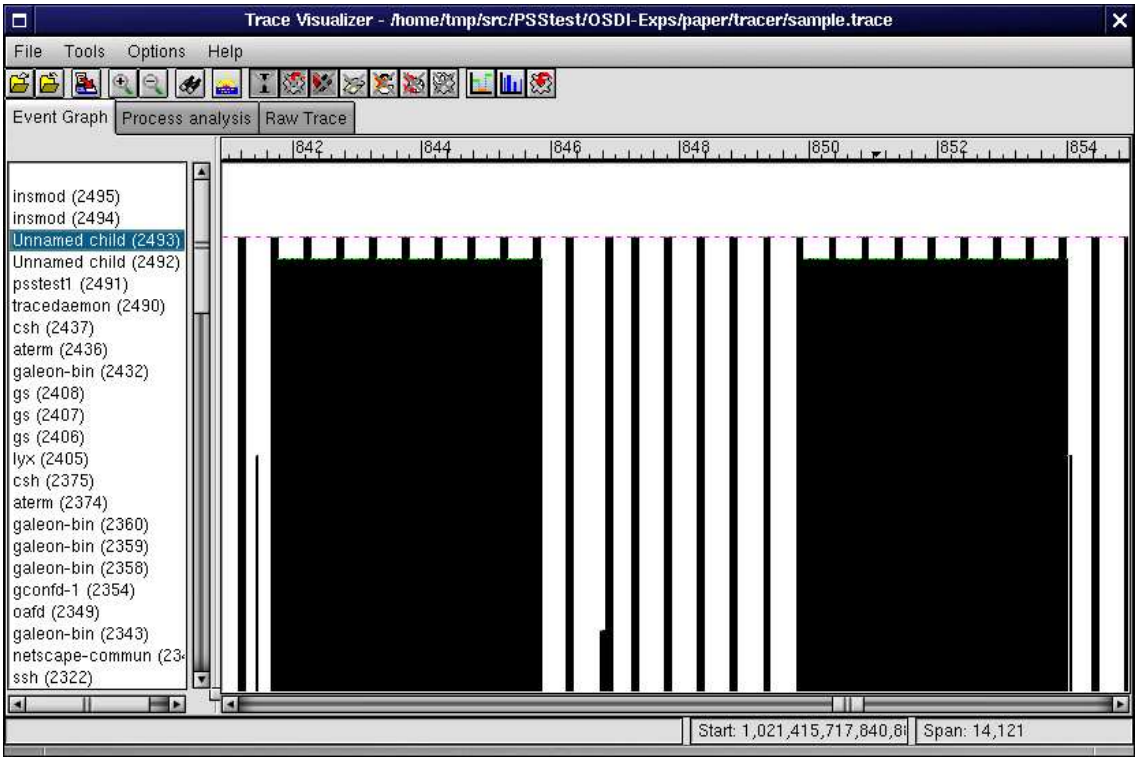
of interrupt processing after real-time processes or explicitly schedule interrupt processing. A less intrusive solution is to compensate for interrupt processing by dynamically allocating CPU to threads based on measured stolen time [93, 3]. We are currently investigating these solutions.

An alternative method for evaluating the scheduler behavior is to use a kernel tracer, such as LTT [113], 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 real-time applications. We ported LTT to TSL. This process was relatively simple since LTT code functionality is orthogonal to TSL functionality. However, LTT code was often added in the same places where firm timers and preemption code had been added. Hence, porting required careful integration. Figure 5.7 shows a sample session analyzing the schedule generated in the previous experiments. All processes in the system are shown on the left. The trace visualizer application shows the process execution schedule on the right with vertical black lines whose height corresponds to the executing processes shown on the left. The two proportion-period processes, marked as "unnamed child" by the trace visualizer and with PIDs 2493 and 2492, are easily recognizable, because they use most of the CPU time. Also note that their execution is regular, and coincides with the expected schedule for two processes with proportion-periods $(40\%, 8192)$ and $(20\%, 512)$. All the non proportion-period processes execute when processes 2492 and 2493 have exhausted their proportions. For example, the `lyx` editor executes for a short period of time after the first period of process 2493.

5.3.2 Real-Rate Controller

This section uses simulation to characterize the performance of our prototype real-rate controller. It examines the responsiveness and stability of the controller as a real-rate thread's resource requirements change over time and explains how the controller parameters should be tuned. The simulation engine is a custom written program. Its implementation closely mimics the controller we have implemented in the Linux kernel except that the simulation uses floating points while the kernel controller performs fixed point arithmetic.

The goal of the controller is to track the processing needs of each thread. This goal can be stated as two sub-goals: 1) minimize delay introduced at the thread and 2) keep the proportion overshoot at a reasonable level, as explained in Section 5.1.3.6. Hence, all results in this section

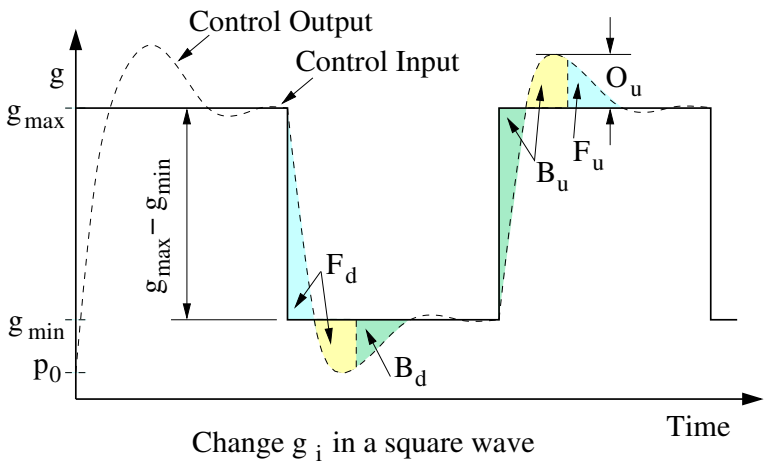


The Linux kernel tracer (LTT) helps to visualize the schedule generated by two proportion-period processes.

Figure 5.7: Linux kernel tracer.

are presented in terms of these two goals. We use simulation to understand the trade-off between these goals and to tune the parameters of the controller. The simulation also provides intuition and a systematic methodology for tuning these parameters.

The processing needs of a thread are modeled by the proportional gain g_i. When this gain increases, then the processing needs of the thread have increased and vice-versa. The basic simulation experiment changes the value of g_i in a square wave and measures the delay and allocation overshoot introduced by the controller, and hence indicates how well the control law together with the estimator tracks g_i. The square-wave signal is a step up and step down signal which shows the effect of changing g_i in the most drastic manner. In practical situations, g_i will vary less dramatically and thus the square wave measures the worst case delay requirements and overshoot due to the control response.



The simulation experiment is performed with a square wave, where the proportional gain g_i is changed from g_{max} to g_{min} and back. The dotted line shows how we expect the controller output to track gain. The areas B_d and B_u show where the thread has accumulated delay while the areas F_d and F_u show where the thread is running ahead. The area B_u is shown twice. In the first area, the thread is accumulating delay because the allocation is lower than the desired g_{max}. In the second area, the allocation is higher and hence the thread catches up. Similarly for the area F_d. The proportion overshoot during the step up phase of the square wave is $O_u/(g_{max} - g_{min})$.

Figure 5.8: Square wave simulation

Figure 5.8 shows how the value of g_i is changed in a square wave. The experiments below measure delay and overshoot of the controller for different values of the step ratio $G = g_{max}/g_{min}$ (i.e., for different instantaneous changes in the mapping between resource requirements and real-rate progress). As the step ratio G increases, the delay and overshoot is expected to increase because the controller takes longer to stabilize to the larger change in the proportional gain. Since $p_i = g_i r_i$ and the goal of the controller is to drive r_i to 1, the output of the controller, p_i, is expected to track g_i. This output is shown with the dashed line in Figure 5.8.

Recall from Equation 5.3 that the delay introduced at a thread by the controller is equal to $iT_s - t_i$. The results below calculate the value of delay by using the maximum positive value of $iT_s - t_i$, or $max(iT_s - t_i)$. In Figure 5.8, this maximum delay is equal to $max(B_d, B_u)$. Note that we do not consider the areas F_d and F_u for delay because here the thread is running ahead.

The allocation overshoot during step up is calculated as $O_u/(g_{max} - g_{min})$, where these values are shown in Figure 5.8. Note that our experiments do not consider proportion undershoot during

step down since reducing the proportion of a thread to less than what it eventually needs does not cause resource overloading. Note also that the simulation starts with the proportion being set to p_0. Initially, the simulation experiments are performed starting with $p_0 = g_0 = g_{max}$ to determine delay and overshoot, when p_0 starts at the correct proportion value. Later, we show how the choice of p_0 affects delay.

The real-rate mechanism has three parameters, α, β and p_0 that need to be tuned to meet the controller's goals. In addition, there are two main factors that affect feedback accuracy: 1) system disturbance, or variations in the proportional gain g_i, the ratio between the proportion of CPU assigned to a thread and the real-rate progress made by the thread, and 2) monitor disturbance, or the granularity of time-stamps T_g generated by a real-rate application. The expectation is that increased variations in g_i and larger value of T_g will reduce feedback accuracy. Our experiments use simulation to model variations in g_i and different T_g. The following sections explain 1) how the parameter α should be chosen, 2) how the parameter β should be chosen, and 3) the effect of the choice of p_0 on delay.

5.3.2.1 Choice of α

The first set of simulation experiments show how the α parameter of the control law in Equation 5.5 should be chosen. To do so, we first show how the choice of α values affects delay and overshoot. Here, time-stamps are assumed to be continuous (i.e., quantization Q is zero or there are no quantization issues due to coarse granularity time-stamps). The next section shows that the optimal value of the estimator parameter β is 1 when quantization is zero. Hence, β is chosen to be 1 in these experiments. In addition, p_0 is chosen to be g_{max} as described earlier. Later sections relax these assumptions.

Figure 5.9 shows the overshoot as α is changed from 0 to 2 when $G = 2$.[5] As expected, larger values of α increase the proportion overshoot because the controller becomes more responsive to variations in current delay. Increasing overshoot can result in overload, which can cause slowing down or suspension of other real-rate applications. Hence, from now on we will choose α values so that overshoot is limited to within a certain maximum range from 10-30% as shown with the

[5]Note that α must lie between these values for stability.

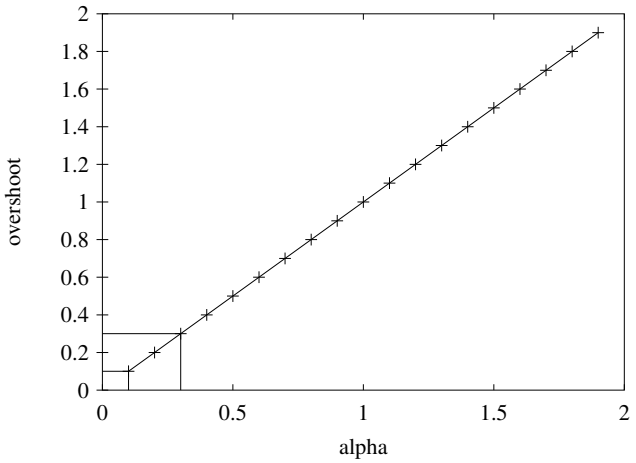


Figure 5.9: Relationship between overshoot and control parameter α.

lines at the bottom-left of Figure 5.9.

Figure 5.10 shows the maximum delay introduced by the controller for different values of the step ratio G. Increasing the value of G implies that the controller has to adapt to larger changes in progress needs, hence the increased delay. Note that the absolute value of the proportion requirements does not affect delay. For example, the increased delay will be the same if the proportion requirements increase from say 10% to 20% versus from 20% to 40%. The real-rate controller possesses this desirable property because it explicitly estimates the value of g_i in Equation 5.4, which it then uses as a parameter for the control law in Equation 5.5. A linear control law that does not estimate g_i but uses a constant parameter would not have this property.

In this graph, the largest possible value of α was chosen to minimize the worst-case delay, while keeping overshoot to less than 15%. In this case, when there is no quantization and β is chosen to be 1, the α value was 0.15 in all cases. Hence, from now on, we will use an α value close to 0.15 because it limits overshoot. In our implementation, α is chosen to be a negative power of 2 for integer arithmetic and thus is 1/8 or 0.125.

5.3.2.2 Choice of β

The previous section described how the control law parameter α should be chosen and showed how delay depends on instantaneous variations in the proportional gain g_i. To simplify the discussion,

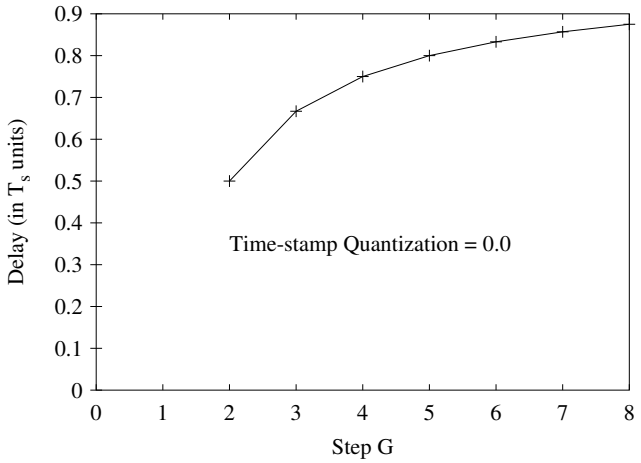


Figure 5.10: Relationship between delay and step ratio G.

it made the unrealistic assumption that time-stamp quantization Q is zero. In practice, real-rate threads perform work and hence progress at a certain granularity. This granularity is captured in the value of Q, which is the ratio of the time-stamp granularity and the sampling period, as described in Equation 5.7. Quantization affects the behavior of the estimator in Equation 5.4. Generally, with increasing quantization, the estimator parameter β must be made smaller to reduce disturbance due to quantization. However, reducing β also reduces the response of the estimator to real changes in the value the proportional gain g_i, which is required by the control law.

This section explores this trade-off with a second set of simulation experiments that vary g_i in a step manner as the previous experiments but do so for different values of quantization Q. These experiments describe how the estimation parameter β should be chosen for Equation 5.4. The choice of β determines the effectiveness with which g_i can be estimated. Since we assume that g_i changes slowly over time, the best choice of β is 1 when time-stamps have no quantization, because then g_i is estimated based on the most current observation. However, if time-stamps have coarse granularity, β should be less than 1 to smooth out the disturbance or the errors caused by quantization.

To understand how β should be chosen for different time-stamp granularities, we performed the previous experiments for measuring delay again. This time α was fixed to $1/8$ while β was

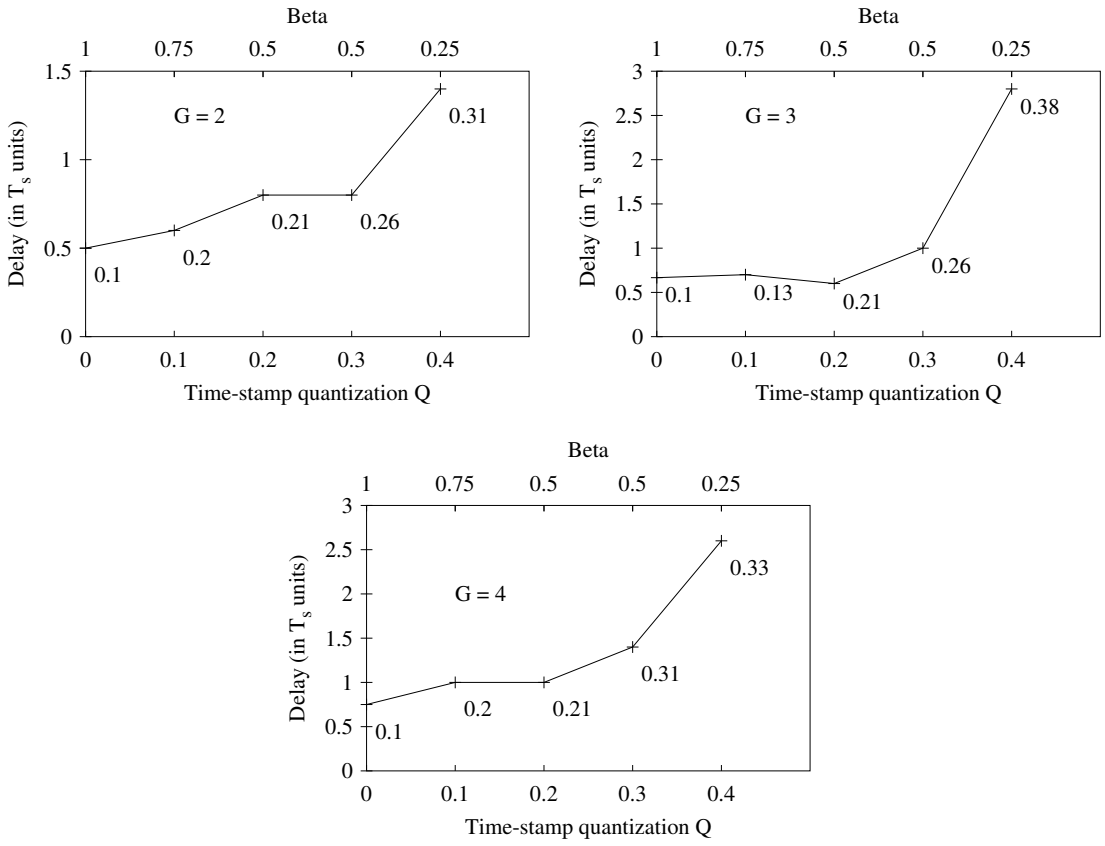
varied. In addition, the quantization Q was varied. Given an accurate time-stamp, the coarse-grained quantized time-stamp was simulated by taking the floor function of the accurate time-stamp with respect to the time-stamp quantization.

Figure 5.11 shows the maximum delay, in sampling period units, introduced by the controller for different values of time-stamp granularity. This figure shows three graphs for three different step ratios $G = 2$, $G = 3$ and $G = 4$. The x-axis of each graph is the time-stamp quantization (i.e., the granularity of time-stamps expressed in sampling period units). These graphs show that increasing granularity increases delay significantly. For example, the delay increases by a factor of more than 2 when the time-stamp quantization changes from 0 to 0.4.

The top of each graph shows the value of β with which each experiment is run. As time-stamp granularity increases, the value of β is decreased because larger granularity implies higher quantization error, which requires more smoothing. At each point in the graph, the overshoot value is also shown. For example, when $G = 2$ and the time-stamp granularity is 0.1, then the overshoot is 0.2 or 20%.

The choice of β in these experiments was made after running several exhaustive experiments with different values of β and the overshoot value. These experiments showed several interesting characteristics. First, with increasing quantization, both the delay and the overshoot values increase, as shown in the graphs. Second, as quantization increases, delay increases exponentially if the overshoot has to be limited to a small value such as 15%. Finally, the values of β shown are close to optimal in terms of delay and overshoot in the following sense. At each quantization, if the β value is larger than the value shown in the graph, then delay decreases by a little but overshoot increases by a large amount because the estimator is too aggressive and does not smooth out the variations in time-stamps. Also, if the β value is smaller than the value shown in the graph, then overshoot decreases by a little but delay increases by a large amount because then the estimator takes too long to respond to actual changes in g_i (i.e., changes due to the step function). The graphs show delay and overshoot for quantization until 0.4. Larger values of quantization cause very large increases in delay or overshoot. The values of β we used (shown in the figure) allow easy integer implementation of the estimator law.

Given the time-stamp granularity of a thread, the graphs in Figure 5.11 provide all the data needed to choose the optimal sampling period of a thread. Such a period would minimize the



These graphs show how the effect of time-stamp quantization on delay. As quantization increases, delay (expressed in sampling period units) increases also. At each point of the graph, the overshoot value is shown, which also increases with increasing quantization. At the top of the graph, the β value that was used for each quantization is shown. These β values are close to optimal for minimizing the maximum delay and overshoot. The three graphs show the delay for the step ratios $G = 2$, $G = 3$ and $G = 4$.

Figure 5.11: Relationship between delay and time-stamp quantization.

worst-case delay experienced by the thread as explained in Section 5.1.3.4. To choose the optimal sampling period, note that the delay in the y-axis in the graphs in Figure 5.11 is expressed in terms of the sampling period. Instead, if we divide each of the delay points by the quantization value, the delay would be expressed in terms of the time-stamp granularities since the quantization $Q = T_g/T_s$ (see Section 5.1.3.4). The three graphs above are combined and shown in Figure 5.12. Here the y-axis shows the delay in terms of any arbitrary time-stamp granularity while the x-axis is still the quantization. This figure shows that the optimal quantization lies between 0.2

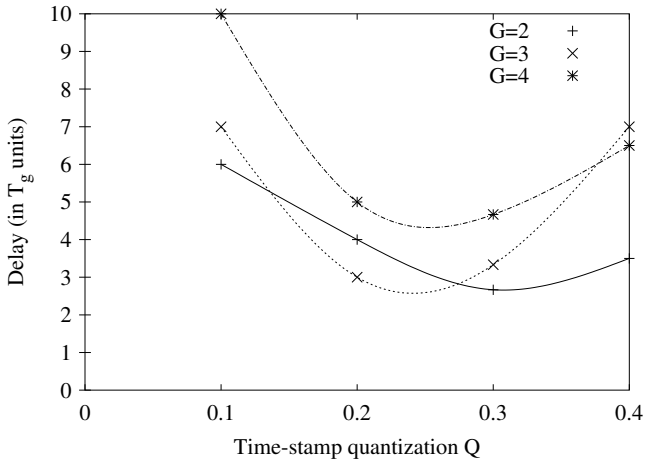


Figure 5.12: Optimal quantization value lies between 0.2 and 0.3.

and 0.3 since then delay in minimal. Based on this figure, the sampling period is optimal when $T_s = T_g/Q_{optimal}$, or the sampling period T_s should be 3 to 5 times the time-stamp granularity.

In summary, Figure 5.12 shows that the real-rate controller should use a sampling period so that the expected quantization in time-stamps is between 0.2 and 0.3. The graphs in Figure 5.11 show that with this quantization, β should be chosen to be 0.5. Section 5.3.3 discusses the implications of these numbers.

5.3.2.3 Effect of Starting Proportion on Delay

The simulation results in the previous section assumed that the proportion p_0 at the start time is chosen to be precisely the correct allocation value (i.e., g_{max} in Figure 5.8). If the value of p_0 is different from g_{max}, startup effects can cause additional delay and larger overshoot as shown in the left side of Figure 5.8.

By carefully examining this figure, it should be clear that, if p_0 is greater than g_{min}, then the delay, which is equal to $max(B_d, B_u)$, will not change because neither of these values will change. However, if p_0 is less than g_{min}, then B_u will be larger (and B_d may be larger) and hence the delay and overshoot will be larger. Note that, when p_0 is greater than g_{max}, then F_d and possibly F_u will be larger but that simply means that the thread is running ahead and hence is not considered part of delay.

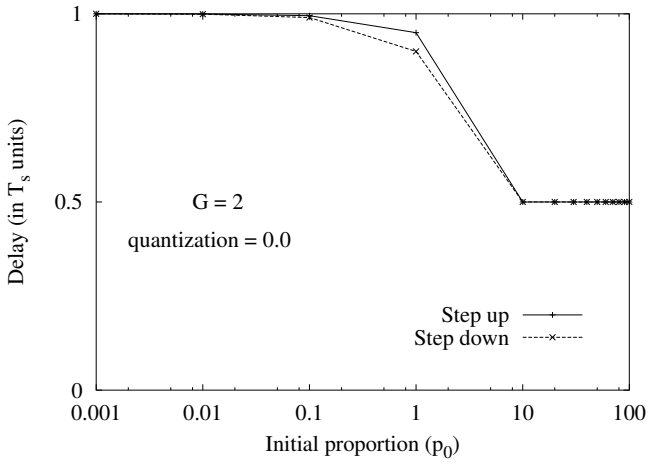


Figure 5.13: Relationship between delay and initial proportion p_0.

To understand how the value of p_0 affects delay, we ran the square wave experiment with different values of p_0. This experiment is run twice, with the square wave first starting at g_{max} and then at g_{min} to capture startup effects in both cases. The value of α is chosen as 1/8 and the value of β is one because time-stamp quantization effects are ignored.

Figure 5.13 shows the results of this experiment. The x-axis in this graph is the initial proportion and is drawn on a log scale. In this experiment, $g_{min} = 10$ (the thread needs 10% of the CPU), $g_{max} = 20$ and hence $G = 2$. As expected, the smallest delay occurs when $g_{min} \leq p_0$. Smaller values of p_0 increase delay until the worst-case delay is twice the minimum worst case delay assuming p_0 was chosen correctly (i.e., $\geq g_{min}$). This figure shows that estimating the initial proportion correctly, such as by caching the proportion during the previous run of the thread, will improve worst-case delay by as much as a factor of 2.

5.3.3 Discussion

Figure 5.12 shows the expected behavior of a real-rate controller. It shows the worst-case delay that can be introduced for different values of instantaneous change in the proportional gain G for any time-stamp granularity. When the sampling period is chosen to be 3 to 5 times the time-stamp granularity ($0.2 \leq Q \leq 0.3$), the delay is minimal and this choice of sampling period is optimal. Under these conditions, if the instantaneous change in g_i is a factor of 2 ($G = 2$), then the expected

worst-case delay lies between 2-3 times the time-stamp granularity.

We made initial measurements of the value of G for some video streams and its value was between 2 and 4 [105]. Assuming $G = 2$, the controller can introduce 66-100 ms of delay for a video application, if the video data is time-stamped 33.3 ms apart. To reduce this delay, data has to be time-stamped at a finer granularity. For example, sub-frames of each frame could be time-stamped at a finer-granularity. A CD audio application with time-stamps 2.27 ms apart (time-stamps every 100 samples in a 44KHz signal) can expect 5-6 ms of delay if the maximum expected variation in G is 2. In the future, we plan to measure the value of G rigorously for other types of real-rate applications and verify the behavior of the controller with respect to the simulation results.

The real-rate controller uses the proportion-period scheduler to schedule applications. We have shown that the proportion-period scheduler is implemented more accurately under TSL. TSL allows fine-grained sampling of progress and actuating of proportions and hence can be used with real-rate threads that specify fine-grained time-stamps for low-delay.

5.4 Conclusions

The key benefit of the real-rate controller is that applications do not have to specify their scheduling requirements in resource specific terms such as CPU cycles. Instead, applications use an application-specific notion of progress such as timing information. The controller uses feedback control to automatically derive the resource requirements based on the timing information. This approach automatically adjusts the allocation of a thread as its resource requirements change over time. For example, we have a multimedia pipeline of processes that communicate with a shared queue. Our controller automatically identifies that one stage of the pipeline has vastly different CPU requirements than the others (the video decoder), even though all the processes have the same priority. This approach allows dependent processes to dynamically achieve stable configurations of CPU sharing that fair-share, weighted fair-share, or priorities do not provide.

This chapter has explained how time-stamps can be monitored from an application, fed to the controller, which then determines the appropriate allocation for the thread. We have shown how the system can be modeled and the controller designed for our non-linear system using standard

feedback linearization techniques. The goal of the controller is to limit proportion overshoot and minimize the delay as experienced by a thread. Our analysis based on simulations has shown how delay can be minimized by using an appropriate sampling period for the controller given the granularity of time-stamps generated by the thread. In general, finer-grained time-stamps and sampling reduces delay. TSL supports such fine-grained sampling, control and scheduling and hence can help minimize scheduling delays.

Our feedback approach uses control analysis and simulation to determine feedback response and stability. While some researchers have used such analysis for feedback scheduling [103], its applicability in generic OS environments has been an open issue since the behavior of software systems is harder to characterize under a variable mix of applications as compared to a dedicated control system. Our real-rate approach shows that software systems can be modeled and analyzed, and further, controllers using standard control techniques can be designed for them using very generic assumptions about the operating environment.

Chapter 6

Tools for Visualization

This chapter describes *gscope*, a visualization tool for low-latency applications. Gscope provides an oscilloscope-like interface that can be integrated with software applications [41]. While Gscope runs on standard Linux, it is itself a low-latency application that can use the fine-grained and accurate timing support available in TSL to poll application variables (signals) at high frequencies and to display the results for visualization.

Gscope focuses on software visualization and is thus designed to handle various types of signal waveforms, periodic or event-driven, in single or multi-threaded environments as well as local or distributed applications. Gscope helps in visually verifying system correctness and modifying system parameters. In our experience, it has been an invaluable debugging and demonstration tool for the low-latency applications we have developed. Our experiments with using gscope show that the library has low overhead.

Section 6.2 explains how gscope benefits from the fine-grained timing available under TSL. Section 6.3 provides an example that shows how we used gscope for visualizing the behavior of MIN_BUF TCP streams. Section 6.4 presents key components of the interface that enable an application to communicate with gscope. Then, Section 6.5 discusses various aspects of programming the gscope library and it describes some of our experiences with gscope. Finally, Section 6.6 presents our conclusions.

6.1 Introduction

Current techniques for visualizing, testing and debugging low-latency applications are long and error prone. They involve some or all of these steps: 1) create an experimental setup, 2) generate

data in real-time, 3) collect data and store it to files, 4) process the file data offline, and 5) plot the data. The first three steps are complicated by the fact that the programmer must attempt to minimize the impact of these steps on the application's timing behavior. The programmer must often repeat these steps several times before being satisfied with the results. In addition, for a distributed application, data files must be collected from multiple machines and transferred to a single machine where the data is correlated before it can be processed.[1] This approach is error prone because the steps outlined above are often not an integral part of the application. Further, with this approach, it is not easy to demonstrate or experimentally validate system behavior in real-time.

We have implemented a software visualization tool and library called *gscope* that borrows some of its ideas from an oscilloscope. The gscope design is motivated by these following goals. First, gscope should simplify visualization and modification of system behavior in real-time, especially the interactions among concurrent or competing software components, within or across machine boundaries. Second, it should enable building compelling software demonstrations that can help explain the internal working of a low-latency system and allow visual verification of system correctness. Third, it should be an easy to use library that complements standard debugging techniques with a real-time "debugging" tool and encourages the use of visualization as an integral part of the application. Finally, it should be a generic and extensible library that does not need specific hardware for correct operation. A key reason for implementing oscilloscopes in dedicated hardware is because they can provide fine-grained and precise timing. We believe that TSL helps to satisfy a software oscilloscope's timing requirements and hence gscope does not require dedicated hardware. This issue is discussed in more detail in Section 6.5.5.

From an ease of use perspective, the oscilloscope interface is ideal. The probes of the oscilloscope are hooked to a circuit and, loosely speaking, the oscilloscope is ready for use. Our goal is to emulate this simplicity in interface as much as possible while extending it when needed to accommodate software needs. In the simplest case, a gscope signal consists of a signal name and a word of memory whose value is polled and displayed. More complex signals consist of functions that return a signal sampling point. The next section describes the polling mechanism in gscope.

[1] In some cases, it is almost impossible to correlate distributed data for analysis, but we will assume that distributed data can be correlated.

In our experience, perhaps the most significant difference between the signals produced by software components and the signals typically visualized in an oscilloscope is the number of signal or event sources. Since software signals are not necessarily tied to specific pieces of hardware, applications can generate large numbers of disparate signals that need to be visualized and correlated. For instance, we use gscope to view dynamically changing process proportions as assigned by the feedback-based proportion-period scheduler (described in Section 5.1). Here, the number of signals depends on the number of running processes. As another example, since software signals are disconnected from hardware, they may be generated from remote sources (see Section 6.5.4).

6.2 Polling in Gscope

Gscope uses the GTK timeout mechanism to implement polling. The default GTK timeout implementation uses the timeout feature of the POSIX `select` call. Although `select` allows specifying the timeout with a microsecond granularity, the standard Linux kernel wakes processes at the granularity of the normal timer interrupt which has a much coarser granularity such as 10 ms on standard Linux.[2] Thus gscope on standard Linux is limited to this polling interval and has a maximum frequency is 100 Hz.

However, gscope on Time-Sensitive Linux allows much higher polling frequency because the `select` implementation in TSL uses firm timers (see Chapter 2) and firm timers have 2-5 μs timer resolution, which is close to the interrupt service time. (see the relevant discussion in Section 3.3.1.1). In practice, the screen pixel size and the screen width can become a limiting factor for display because if the polling frequency is very high and the display shows each polled sample at the next pixel, then the signal will sweep the display very quickly and will not be easily visible. For example, if one second worth of samples should be visible on the screen at a given time and the screen size is 1600 pixels across, then the sampling period is limited to 625 μs. Note that the display refresh rate is not a limiting factor at a high polling frequency because the samples will still be displayed, although the display will batch several samples during refresh. To solve the screen size problem, samples can be recorded at a high frequency and then displayed in playback mode later (see Section 6.4.1).

[2]The `setitimer` periodic timer call behaves similarly on standard Linux.

Gscope is itself a low-latency application and thus preemption and scheduling latencies in the kernel can induce loss in polling timeouts under heavy loads. Under TSL, preemption latencies are almost always less than 20 μs (as shown in Figure 3.6). In addition, to reduce scheduling latency, gscope can be run as a high-priority real-time application. In case lost timeouts occur, gscope advances the scope refresh appropriately without displaying the trace.

6.3 A Gscope Example

This section describes how we used the gscope library is used to visualize network behavior for MIN_BUF TCP flows. We used the `Mxtraf` traffic generator [63] that incorporates gscope to induce traffic in an experimental network. The experiment shown in Figure 6.1 shows the behavior of TCP flows in a relatively congested wide-area network. In this experiment, we generate a varying number of long-lived flows (called *elephants*) that transfer data from the server to the client. This figure shows two signals. The `elephants` signal shows the number of long-lived flows over time (the y-axis). This number is changed from 10 to 20 roughly half way through the x-axis (i.e., 10 seconds into the experiment). The `Cwnd` signal shows the TCP congestion window (at the server) of one (arbitrarily chosen) long-lived flow. This window provides an estimate of the short-term bandwidth achieved by the flow.

This figure shows how the congestion window changes with a changing number of long-lived flows. The first obvious thing to notice is that the congestion window size is reduced with increasing number of flows. The lowest value of the `Cwnd` signal in the graphs corresponds to a `Cwnd` value of one. TCP reduces its congestion window to one upon a timeout. The figure shows that this value is only reached when the number of flows is 20 and not when it is 10. Additional signals (not shown in these figures for simplicity) confirm that there is a timeout each time `Cwnd` reaches one. These timeouts affect TCP throughput and latency significantly.

Visualization using gscope has revealed several interesting properties in TCP behavior that would have been hard to determine otherwise. For instance, our initial MIN_BUF TCP implementation showed significant unexpected timeouts that we finally traced to an interaction with the SACK implementation and which lead us to implement the sack correction described in Section 4.4.

Figure 6.1: A snapshot of the `GtkScope` widget showing TCP behavior

6.4 Gscope API

This section describes the interface data structures that enable an application to communicate with the gscope library. The gscope interface is relatively simple but powerful and consists of three components: 1) signal specification, 2) control parameter specification for configuring the application, and 3) tuple format for streaming signals in real time and for recording and viewing data offline.

Gscope has been implemented using the Gnome [25] and GTK [44] graphical toolkits. These GUI toolkits are multi-platform although they are primarily designed for the X Window System. Both Gnome and GTK use the Glib library that provides generic system functionality independent of the GUI. For instance, Glib provides portable support for event sources, threads, and file and socket I/O. Gscope uses some of this Glib functionality.

The main graphical widget in the gscope library is called `GtkScope`, as shown in Figure 6.2. An application creates a signal by making a `GtkScopeSig` data structure for each signal that encodes properties of the signal. Then it passes this data structure to the gscope library for display.

The library creates a `GtkScopeSignal` object for each signal. Applications can create one or more `GtkScope` widgets and one or more signals in each scope. A screen shot of the `GtkScope` widget with the embedded canvas displaying two signals was shown in Figure 6.1.

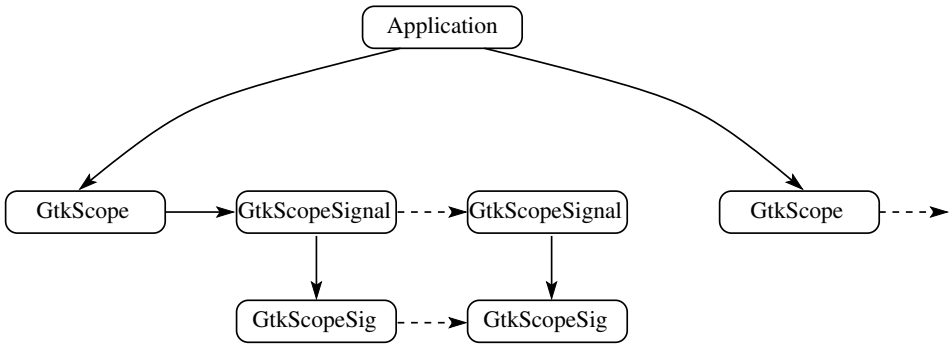


Figure 6.2: The `GtkScope` widget

6.4.1 Signal Interface

Gscope can acquire signal data from applications in one of two acquisition modes: *polling* or *playback*. In polling mode, signals are obtained from the running program using the signal interface described below. Polled signals can be *unbuffered* or *buffered*. In unbuffered mode, gscope polls and displays single sampling points. In buffered mode, applications enqueue signal samples with time-stamps into a buffer and gscope displays these samples with a user-specified delay. The buffered mode enables applications to push data to the scope. For instance, an application can listen for kernel events on a `netlink` socket and push these event samples to the gscope buffer. Gscope polls the buffer periodically to display the samples. Polled signals can be displayed in the time or frequency domain. In addition, the polled data can be recorded to a file. Section 6.5 discusses the polling overhead and the finest polling granularity that is supported in gscope.

In the playback mode, data is obtained from a file and displayed. This file format is described in Section 6.4.3. Both polling and playback modes have a polling period associated with them. In both modes, data is displayed one pixel apart each polling period (for the default zoom value).

A signal is specified to the gscope library using a `GtkScopeSig` structure shown below:

```
typedef struct {
```

```
  char             *name;   /* signal name */
  GtkScopeSigData   signal; /* signal data */
  /* color, min, max, line, hidden, filter */
} GtkScopeSig;
```

The `name` is the name of the signal and the `signal` field is used to obtain signal data. This field is described with an example below. The rest of the fields are optional parameters that specify the color of the signal, the minimum and maximum value of the signal displayed (for default zoom and bias values), the line mode in which the signal is displayed, whether the signal is hidden or visible, and a parameter α for low-pass filtering the signal. The low-pass filter uses the following equation to filter the signal: $y_i = \alpha y_{i-1} + (1 - \alpha)x_i$. Here, x_i is the signal point and y_i is the filtered signal point. The α filter parameter ranges from the default value of zero (unfiltered signal) to one.

The examples below show the `GtkScopeSig` specification for the `elephants` and `Cwnd` signals. The `elephants` signal consists of an integer value that will be sampled by gscope. The `Cwnd` signal uses the `get_cwnd` function to determine the `Cwnd` value of the socket `fd`.

```
int elephants;
GtkScopeSig elephants_sig = {
  name:   "elephants",
  signal:  {type: INTEGER, {i: &elephants}},
  min:     0, max: 40  /* optional */
};

int fd;   /* socket file descriptor */
GtkScopeSig cwnd_sig = {
  name: "Cwnd",
  signal: {type: FUNC, {fn: {get_cwnd, fd}}},
};
```

The signal can be of type INTEGER, BOOLEAN, SHORT, FLOAT, FUNC or BUFFER and this type determines how signals are sampled. When the signal type is BUFFER, the signal is buffered, otherwise it is unbuffered.

Unbuffered Signals

For unbuffered signals, the INTEGER, BOOLEAN, SHORT, FLOAT field in the `GtkScopeSigData` union is sampled and displayed, depending on the `type` of the signal. When the signal type is `FUNC`, the `fn` function field in the union is invoked with the two arguments `arg1` and `arg2` (passed in by the user during `GtkScopeSig` initialization) and the function's return value is the value of the signal data. The function mechanism allows reading arbitrary signal data.

Buffered Signals

For buffered signals, gscope reads data from a scope-wide buffer that has time-stamped signal data in a tuple format (described in Section 6.4.3) and displays this data with a user-specified delay. Gscope provides applications an API for inserting the time-stamped signal data in the buffer. Currently, unbuffered signals are never delayed but, in the future, it may be useful to delay such signals so that they can be compared or matched with buffered signals.

6.4.2 Control Parameter Interface

The `GtkScopeParameter` structure in the gscope library can be used to read and modify application parameters. These parameters are not displayed but generally used to modify application behavior. The `GtkScopeParameter` structure is very similar to the `GtkScopeSig` structure. However, while signals can only be read, application parameters can also be modified.

6.4.3 Tuple Format

Signals can be streamed to gscope. For instance, streamed signals allow visualization across machines in real time. Signals can also be recorded to a file and gscope can replay signals from the file. In all these cases, signal data is delivered, generated or stored in a textual tuple format. Each tuple consists of three quantities: *time*, *value* and *signal name*. This format allows multiple signals to be delivered to gscope or recorded in the same file. As a special case, if there is only one signal, then the third quantity may not exist. In that case, signals are simply time-value tuples.

When signals are replayed from a recorded file, the time field of successive tuples should be in increasing time order and its value is in milliseconds. Data is displayed one pixel apart for each

polling period (for the default zoom value). For instance, if the polling period is 50 ms, then data points in the file that are 100 ms apart will be displayed 2 pixels apart.

6.4.4 Programming With Gscope

The gscope library has a programmatic interface for every action that can be performed from the GUI. Figure 6.3 presents a fragment of a simple program that shows how the gscope library is used. After creating `scope`, the `elephants_sig` signal (defined in Section 6.4.1) is added to `scope` and `scope` is set to polling mode, where it polls the value of `elephants` every 50 ms. The client changes the value of elephants and sends a message to the server on the control socket `fd`. The server reads the value of elephants in the function `read_program, which runs` when the control data becomes available from the client. In this usage style, the `read_program` function is I/O driven and performs non-blocking calls. Other ways of using the gscope library include 1) periodic invocation of `read_program` and 2) separation of the scope into its own thread. These issues are discussed further in Section 6.5.3.

6.5 Discussion

The previous section has described the gscope API and how the gscope library can be used. This section discusses various aspects of programming the gscope library in more detail and it describes some of our experiences with gscope. Section 6.5.1 describes portability issues with the gscope library. Section 6.5.2 examines how gscope can be effectively used for different types of signals. Section 6.5.3 describes when it is appropriate to have a single-threaded or a multi-threaded gscope application, while Section 6.5.4 describes how data is polled and displayed from a distributed application. Section 6.5.5 describes the polling granularity in the current implementation and thus the type and range of applications that can be supported. Finally, Section 6.5.6 discusses the overhead of our approach.

6.5.1 Implementation Portability

Gscope has been implemented on Linux and we have been using it for the last two years. Gscope can be installed on a vanilla Linux system that has Gnome software installed on it. Although

```
main()
{
  ...
  scope = gtk_scope_new(name, width, height);
    /* sig defined in Section 6.4.1 */
  gtk_scope_signal_new(scope, elephants_sig);
    /* sampling period is 50 ms */
  gtk_scope_set_polling_mode(scope, 50);
    /* set polling to start state */
  gtk_scope_start_polling(scope);
    /* register read_program with I/O loop */
  g_io_add_watch(..., G_IO_IN, read_program, fd);
    /* main loop: calls read_program when fd
       has input data */
  gtk_main(); /* doesn't return */
}

gint
read_program(int fd)
{
  control_info = read_control_info(fd);
  if (elephants != control_info.elephants) {
    start or stop elephants;
      /* change signal value */
    elephants = control_info.elephants;
  }
  return TRUE;
}
```

Figure 6.3: A sample gscope program

gscope has not been ported to other operating systems such as BSD, we believe that the porting process should be simple because gscope does not use any Linux specific functionality and because Gnome has been ported to other OSs.

6.5.2 Signal Types

Applications often produce various types of signal data, such as clocked signals and event-driven signals. For instance, bandwidth monitoring can be done based on events that are packet arrivals. Gscope implements a discrete-time polling system but can also handle event-driven signal generation. Below, we describe various signal types and how gscope handles them.

Sample and Hold: Applications can be designed so that certain events change a state and then the state is held until the next event changes the state. Between event arrivals, polling can detect

the previous event by monitoring the held state. For instance, the state can be the end-to-end packet latency that can change on each packet arrival event. If the polling frequency is sufficiently high, all packet arrival events can be captured. This approach requires knowing the shortest period of back-to-back event arrivals.

Periodic Signals: Signals can be periodic, in which case, such signals can be viewed by polling at the same period. For instance, we use gscope to view dynamically changing process proportions as assigned by a real-rate proportion-period scheduler [104]. These proportions are assigned at the granularity of the process period and we set the scope polling period to be the same as the process period. This approach does not require phase alignment between the process period and the polling period since the signal is held between process periods. One issue with this approach is that the polling rate in gscope is per scope and not per signal. If processes have different periods, then the polling rate in the scope should be the greatest common divisor (GCD) of the periods of all the processes.

Buffering: Events can be buffered and then polling can display data with some delay. For instance, a device driver could poll a memory mapped device at the appropriate frequency and queue data to a buffer that is then polled by gscope. In gscope, the buffering interface is implemented with buffered signals described in Section 6.4.1. This approach may seem like cheating since one of the main purposes of a scope is to poll the data directly. However, decoupling the data collection from the data display has several benefits. For instance, data can be collected and displayed on separate machines, thus allowing distributed or client-server application data visualization as discussed in Section 6.5.4. In addition, data can be captured only when certain conditions are triggered (i.e., at "interesting" times). At other times, data collection and display would have little or no overhead. A similar trigger-driven sampling approach is used by hardware oscilloscopes.

Event Aggregation: Another very effective method for visualizing event-driven signals is event aggregation. In this method, event data is aggregated between polling intervals and then displayed. For instance, applications may want to display the maximum value of an event sample between polling intervals. An example of using the maximum sample value is to display the maximum latency of a network connection. Rather than displaying the latency

of each packet (as discussed in Sample and Hold) or the maximum latency over the life time of the connection, it may be useful to display the maximum latency within each polling interval. Gscope provides aggregation functions shown below that aggregate data between polling intervals. Examples for network connections are described for each function.

Maximum and Minimum: maximum and minimum sample, e.g., latency.

Sum: Sum of the sample values, e.g., bytes received.

Rate: Ratio of the sum of sample values to the polling period, e.g., bandwidth in bytes per second.

Average: Ratio of the sum of sample values to the number of events, e.g., bytes per packet.

Events: Number of events, e.g., number of packets.

Any Event: Did an event occur between polling intervals, e.g., any packet arrived?

6.5.3 Single versus Multi-Threaded Applications

Gscope is thread-safe and can be used by both single-threaded and multi-threaded applications. With multi-threaded applications, typically gscope is run in its own thread while the application that is generating signals is run in a separate thread. This approach allows the gscope GUI to be scheduled independently of the application (unless gscope signals make application calls that need to acquire locks). However, it is the application thread's responsibility to acquire a global GTK lock if it needs to make gscope API calls.

Single-threaded gscope applications must use event-driven programming. Such applications should either be periodic or they should be I/O driven and they should use non-blocking I/O system calls (since blocking calls would block the GUI as well). Periodic applications are supported directly by gscope. I/O driven applications can use the GTK `GIOChannel` functions to drive their events as shown in Figure 6.3. This approach allows all GUI and application events to be handled by the same event loop and does not require any locking. However, application logic can become more complex due to the use of non-blocking I/O system calls. We have implemented a single-threaded I/O driven gscope client-server library that is described in the next section.

6.5.4 Distributed Applications

Gscope supports monitoring and visualization of distributed applications. It implements a single-threaded I/O driven client-server library that can be used by applications to monitor remote data. Clients use the gscope client API to connect to a server that uses the gscope server library. Clients asynchronously send BUFFER signal data in tuple format (described in Section 6.4.3) to the server. The server receives data from one or more clients asynchronously and buffers the data. It then displays these BUFFER signals to one or more scopes with a user-specified delay, as described in Section 6.4.1. Data arriving at the server after this delay is not buffered but dropped immediately.

Currently, we use the gscope client-server library in the `mxtraf` network traffic generator. The gscope client-server library allows visualizing and correlating client, server and network behavior (connections per second, connection errors per second, network throughput, latency, etc.) within a single scope.

6.5.5 Polling Granularity

Compared to an oscilloscope, gscope has a coarser polling granularity and thus lower bandwidth. For instance, the current gscope implementation may not be appropriate for real-time low-delay display of speech frequency in a speech recognition application that monitors phone-line quality 8 KHz audio signals (0.125 ms sampling period) during heavy system load because it could occasionally be interrupted due to long preemption latency in the kernel. Fortunately, in our experience, many software applications don't have very tight polling requirements. Coarse granularity polling works well for three reasons: 1) many software applications have coarse time scales, 2) debugging software applications often only requires visualizing the long term trends of the signal, 3) many applications generate event-driven signals that are handled by techniques such as buffering or event aggregation as explained in Section 6.5.2. For instance, the audio signal could be read from the audio device and buffered by an application and gscope can display the signal with some delay using buffered signals. In our experience, a one millisecond polling granularity has been sufficient for most gscope applications that we have implemented on TSL.

6.5.6 Scope Overhead

We measured the overhead of using the gscope library by running a simple application that polls and displays several different integer values. To measure overhead, we use a CPU load program that runs in a tight loop at a low priority and measures the number of loop iterations it can perform at any given period. The ratio of the iteration count when running gscope versus on an idle system gives an estimate of the gscope overhead.

We measured the overhead on a 600 MHz Pentium III processor at 4 different sampling periods: 50 ms (20 samples/sec), 10 ms (100 samples/sec), 5 ms (200 samples/sec) and 1 ms (1000 samples/sec). As expected, the gscope CPU overhead increases linearly with polling frequency. The gscope overhead is 0.76%, 3.1%, 5.9% and 28.8% at these 4 sampling periods. The linear increase is roughly 2.8% for every additional 100 samples per second. Almost all the time in gscope is spent in the X server to display the data.

The increase in overhead with increasing number of signals being displayed ranges from 0.02 to 0.05 percent per signal. When compared to the number of signals displayed, polling granularity has a much larger effect on CPU consumption.

6.6 Conclusions

Gscope focuses on visualization of low-latency software applications. It can be used for visualizing time-dependent variables such as network bandwidth, latency, jitter, fill levels of buffers in a pipeline, CPU utilization, etc. We have implemented gscope and have been using it for the last two years. We have used it for visualizing and debugging various low-latency applications, including a CPU scheduler [104], a quality-adaptive streaming media player [64], a network traffic generator called `mxtraf` [63], and various control algorithms such as a software implementation of a phase-lock loop [36]. We believe that applications using gscope will see a direct benefit in terms of reducing the visualizing, debugging and testing cycle time.

Chapter 7

Related Work

This chapter provides background information related to this research. For organization, the subsequent sections are structured according to the areas of interest in this thesis. Hence, Section 7.1 describes related work in timing control in OSs, while Section 7.2 presents kernel preemption techniques used in other OSs. Section 7.3 discusses OS, application-level and network techniques for managing buffers, especially output-buffering and network latency. The related work on CPU scheduling techniques is presented in Section 7.4. Finally, Section 7.5 describes related work in visualizing and debugging low-latency applications.

While many of the challenges for reducing latency in the OS have been addressed in the past, they have generally not been integrated in general-purpose OSs. When applied in isolation, they fail to provide low latencies. For example, a high resolution timer mechanism is not useful to user-level applications without a responsive kernel. This probably explains why soft timers [9] did not export their functionality to the user level through the standard POSIX API. Conversely, a responsive kernel without accurate timing such as the low-latency Linux kernel [79] provides low latency only to data-driven applications where 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 low-latency 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, while 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 [78]. Real-time operating

systems integrate these solutions to support certain kinds of low-latency applications but tend to ignore the performance overhead of their solutions on throughput-oriented applications [87].

7.1 Timing Control in OSs

An accurate timing mechanism is crucial for supporting low-latency applications. There are two aspects to timing control. First, applications should be able to get the current time accurately and with low overhead. Second, applications should be able to schedule operations at (or sleep until) a precise time in the future with low overhead.

Modern hardware provides facilities that help in obtaining the current time cheaply. For example, modern Intel Pentium machines provide the `rdtsc` instruction that allows reading the CPU cycle counter at the user level in a few cycles. These measurements have low overhead and are very accurate. The latest version of Linux (in development) provides fast user-level library calls that return the current time of day by using this instruction. These calls do not have to make system calls because the relevant timing-related information in the kernel is mapped to user space (in read-only mode). Interestingly, fast and accurate availability of current time allows inferring scheduling behavior at the user level. For example, Hourglass [92] is a user-level application that repeatedly calls the `gettimeofday()` system call to get current time and infers scheduling anomalies by measuring the time difference between the returned values.

Modern machines, such as Pentium II and higher Intel x86 machines, also provide cheap and accurate one-shot timer interrupts. These interrupts can be used to implement high resolution timers. 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 [98] and has subsequently been used by several other systems such as Rialto [59] and, RED Linux [115]. 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 [9]. Thus, our firm-timer implementation incorporates soft timers.

Current OSs often use NTP [77], which is a distributed protocol for synchronizing times across machines. In our experiments, we measure end-to-end latency between a sender and a receiver by

comparing times across machines. We rely on NTP for time synchronization.

7.2 Kernel Preemptibility

The advantages of kernel preemptibility are well-known in the real-time community [76]. However, general-purpose OSs have tended to ignore this issue because they did not focus on low-latency applications. Recently, kernel preemptibility techniques have been incorporated in several different systems. For example, RED Linux [115] inserts preemption points in the kernel, and Timesys Linux/RT (based on RK technology [87]) uses kernel preemptibility for reducing preemption latency. Kernel preemptibility is also used by MontaVista Linux [78].

Recently, there has been renewed interest in the evaluation of these latency reduction techniques. Concurrent with our work (and unknown to us), Clark Williams from Red Hat [111] evaluated scheduling latency in Linux in a manner similar to the one presented in this thesis. The main difference is that Williams uses a decomposition of input latency that is different from ours and he does not explicitly consider timer resolution latency. Williams comes to conclusions similar to ours, although his numerical results are slightly different from our results. One probable reason for this discrepancy is that he uses a different version of the Linux kernel (and the kernel patches that implement different preemption techniques) for the evaluation and he did not use the lock-breaking version of Linux. We are still investigating how his numbers relate to our results.

7.3 Buffering for I/O

This thesis shows that output-buffering latency can be reduced significantly for TCP streams by adaptively sizing the send buffer so that the buffer only contains TCP packets that have to be retransmitted. We implemented this technique in the context of TCP because we believe that a standard congestion-controlled transport protocol such as TCP should be used for low-latency streaming applications.

7.3.1 TCP-based Streaming

The feasibility of TCP-based stored media streaming has been studied by several researchers. Generally, the trade-off in these QoS adaptive approaches is short-term improvement in video

quality versus long term smoothing of quality. Rejaie [94] uses layered video and adds or drops video stream layers to perform long-term coarse grained adaptation, while using a TCP-friendly congestion control mechanism to react to congestion on short-time scales. Krasic [64] contends that new compression practices and reduced storage costs make TCP a viable and attractive basis for streaming stored content and uses standard TCP, instead of a TCP-friendly scheme, for media streaming. Feng [31] and Krasic use priority-based streaming, which allows a simpler and more flexible implementation of QoS adaptation. We believe that similar QoS adaptive approaches will be useful for low latency streaming also.

Researchers in the multimedia and networking community have proposed several alternatives to TCP for media streaming [106, 11, 33]. These alternatives aim to provide TCP-friendly congestion control for media streams without providing reliable data delivery and thus avoid the latency introduced by packet retransmissions. Unfortunately, the effects of packet loss on media streaming can be severe. For instance, loss of the header bits of an *I*-frame in an MPEG movie can render a large segment of surrounding video data unviewable. Thus media applications over a lossy transport protocol have to implement complex recovery strategies such as FEC [96] that potentially have high bandwidth and processing overhead. The benefit of FEC schemes for loss recovery is that they often have lower latency overhead as compared to ARQ schemes such as employed in TCP. Thus, Nonnenmacher [86] explores introducing FEC as a transparent layer under an ARQ scheme to improve transmission efficiency.

7.3.2 Buffer Tuning

Our send-buffer adaptation approach is similar to the buffer tuning work by Semke [99]. However, unlike their work which focuses on improving TCP throughput, our work focuses on reducing socket to socket latency. Semke tunes the send buffer size to between $2*$CWND and $4*$CWND to improve the throughput of a high bandwidth-delay connection that is otherwise limited by the send buffer size. The $4*$CWND value is chosen to limit small, periodic fluctuations in buffer size. Our work shows that a connection can achieve throughput close to TCP throughput by keeping the send buffer size slightly larger than CWND and also achieve significant reduction in buffering latency.

7.3.3 Media Streaming Applications

Popular interactive streaming applications include Voice over IP (VoIP) products such as Microsoft NetMeeting [50]. NetMeeting provides reasonable voice quality over a best effort network but is implemented over UDP because the delays introduced by TCP are considered unacceptable. This thesis shows that adaptive tuning of TCP buffers should yield acceptable delays, especially for QoS adaptive applications. For interactive applications, ITU G.114 [51] recommends 150 ms as the upper limit for one-way delay for most applications, 150 to 400 ms as potentially tolerable, and above 400 ms as generally unacceptable delay. The one way delay tolerance for video conferencing is in a similar range, 200 to 300 ms.

The response time of control operations for media streaming depends on several other factors such as disk I/O and application latency. Dey [26] considers client latency for restarting playback operations (i.e., the time it takes to restart viewing video after data arrives at the client). Chang [18] focuses on disk /IO and reduces initial startup latency with careful data placement and disk scheduling. Similarly, Reddy [91] describes a video server that schedules disk I/O based on urgent and non-urgent requests. Urgent requests are used for restarting playback and stream surfing. These schemes complement our work.

7.3.4 Network Infrastructure for Low-Latency Streaming

Many differentiated network services have been proposed for low latency streaming. These schemes are complementary to our work since, a adaptively-sized send-buffer TCP implementation can be used for the low delay flow. Expedited Forwarding [54] aims to provide extremely low loss and low queuing delay guarantees. Dovrolis [27] describes a proportional differentiation model that allows tuning per-hop QoS ratios between classes to achieve differentiation in queuing delays at intermediate routers. Nandagopal [81] proposes a core stateless QoS architecture which provides per-class per-hop average delay differentiation and class adaptation to maintain desired end-to-end average delay. Hurley [48] provides a low-delay alternative best-effort (ABE) service that trades high throughput for low delay. The ABE service drops packets in the network if the packets are delayed beyond their delay constraint. In this model, the client must recover from randomly dropped packets. Further, unlike with TCP, the server does not easily get back-pressure

feedback information from the network in order to make informed QoS adaptation decisions.

Active queue management and explicit congestion notification (ECN) [90] have been proposed for improving the packet loss rates of TCP flows. Salim [97] shows ECN has increasing throughput advantage with increasing congestion levels and ECN flows have hardly any retransmissions. Feng [19] shows that adaptive active queue management algorithms (Adaptive RED) and more conservative end-host mechanisms can significantly reduce loss rates across congested links. Bagal [10] also shows low packet losses with ECN and, in addition, shows that explicit TCP rate control reduces variance in throughput. These mechanisms should help reduce spikes in protocol latency.

Claffy [21] presents the results of a measurement study of the T1 NSFNET backbone and delay statistics. In 1992, the one way median delays between end points ranges from 20 to 80 ms with a peak at 45 ms. Newer data in 2001 [47] shows that the median round-trip time for East-coast to East-coast or West-coast to West-coast is 25-50 ms and East-coast to West-coast is about 100 ms. We used these median results in our TCP experiments. US to Europe median RTT is currently 200 ms. While the 200 ms median RTT makes interactive applications challenging, responsive control operations for streaming media should be possible.

7.4 CPU Scheduling

Current general-purpose OSs provide a simple virtual machine API to applications. Thus, each application executes on this virtual machine assuming no other applications are present. For example, each application is provided with a contiguous area of (virtual) memory even though this memory is physically disjoint. Although this virtual machine API is simple and easy to use, it does not provide timeliness properties. For example, applications have little control over when and how often they will be executing.

7.4.1 Real-Time Scheduling Algorithms

The real-time community has studied the scheduling problem extensively [67, 75, 101]. There are two classes of real-time algorithms: priority-based and proportion-period algorithms. A key concern with real-time scheduling schemes is analysis of thread schedulability (i.e., what conditions

are necessary so that threads do not miss their deadlines). For example, it is known that for thread schedulability under EDF, the sum total of the requested resources should be less than or equal to 100%.

For this analysis, these algorithms make several simplifying assumptions about application needs. For example, they assume that threads have periodic deadlines and require constant execution time within each deadline period. They also assume that threads yields the CPU voluntarily and the system is fully preemptible. In addition, they assume that threads are *independent* of each other (i.e., the execution of one thread does not depend on the execution of other threads).

Priority-based Scheduling

With priority-based scheduling, the scheduler assigns real-time priorities to threads based on application needs [67]. These needs are specified in the form of execution time requirements and deadlines. Then the scheduler executes the highest priority runnable thread at any instant of time. The scheduler can assign priorities to threads either statically or dynamically. Two classic priority-based algorithms are rate monotonic (RM) scheduling which uses static priorities and earliest deadline first (EDF) scheduling, which uses dynamic priorities.

Rate monotonic scheduling assumes that threads have periodic deadlines and statically assigns priorities to threads such that threads with smaller periods have higher priorities. Thus the thread with the smallest period is given the highest priority. Liu and Layland's seminal paper on RM scheduling [67] shows schedulability can only be guaranteed for large task sets when their CPU requirements are less than 69%.

The earliest-deadline-first scheduler dynamically assigns priorities to threads in their deadline order. The thread with the closest deadline is assigned the highest priority while the thread with the farthest deadline is assigned the lowest priority. Liu and Layland [67] show that the processor can be fully utilized with EDF scheduling (i.e., there is no processor idle time prior to a missed deadline).

Although the EDF scheduler is more efficient than the RM scheduler, it has two drawbacks. First, it is more expensive to implement because, unlike RM scheduling, it has to maintain an additional queue sorted by deadline order. Second, if threads do not satisfy the assumptions described above and miss deadlines, then the thread order in which deadlines are missed is non-deterministic

unlike with RM scheduling. For example, with RM scheduling, threads with longer periods always miss deadlines before threads with shorter periods. EDF schedulers offer no similar property when deadlines are missed and thus reasoning about overload behavior is harder.

Proportion-Period Scheduling

Priority-based scheduling techniques always execute the highest priority runnable process. However, with this solution, a misbehaving high-priority thread that does not yield the CPU can starve all other lower-priority threads in the system. Ideally, an OS should provide *temporal protection* to threads so that misbehaved threads that consume "too much" execution time do not affect the schedule of other threads. The temporal protection property is similar to memory protection in standard operating systems that provides memory isolation to each application.

The proportion-period allocation model automatically provides temporal protection because each thread is allocated a fixed proportion every period. Hence, this model provides fine-grained control over resource allocation and avoids accidental starvation and denial-of-service attacks. The proportion of a thread is the amount of CPU allocation required every period for correct thread execution. The period of a thread is related to some application-level delay requirement of the application, such as the period of a periodic thread, or it can be derived from the jitter requirements of a low-latency application. Intuitively, the period defines a repeating deadline. To meet the deadline, the application must perform some amount of work. Hence, to satisfy the application the scheduler must allocate sufficient CPU cycles in each period. If the scheduler cannot allocate the appropriate amount of time to the thread, the thread is said to have missed a deadline.

7.4.2 Real-Time Schedulers in General-Purpose OSs

Real-time systems provide an API that allows applications to express their timing constraints to the OS and the OS provides fine-grained execution control and allows analysis of thread schedulability. 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 implements Resource Reservations in the

Linux kernel, and RED Linux provides a generic scheduling framework for implementing different real-time scheduling algorithms. These kernels tackle the practical systems issues mentioned above with preemptibility techniques similar to the techniques presented in this thesis. However, while these kernels work well for certain low-latency 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 [12] and RTAI [69], 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 threads are not Linux processes, but run on the lower-level real-time executive, and the Linux kernel runs as a non real-time thread. This solution provides good real-time performance, but does not provide it to Linux applications. Linux processes are still non real-time, hence we believe that RTLinux-like solutions are not usable for supporting low-latency applications running in user space.

Traditionally, Linux, Solaris, and NT provide "real-time" priorities, which are fixed priorities that are higher in priority than priorities allocated to conventional throughput-oriented threads. This priority-based approach does not provide fine-grained control over resource allocation because priorities cannot be directly derived from or represent resource needs. Scheduling based on proportion and/or period provides more fine-grained control over resources. Thus, several proportional share scheduling mechanisms have been implemented [110, 108, 43, 58, 82, 114, 107, 60] in the FreeBSD, Linux, Solaris and Windows kernels and DSRT [45] is a user-level scheduling solution. All these approaches require human experts to supply accurate specifications of proportion and/or period, and focus on how to satisfy these specifications in the best way. None of them try to infer the correct proportion, or adapt dynamically to changing resource needs of the applications.

7.4.3 Hybrid Scheduling

Several systems use hybrid approaches to merge the benefits of reservation-based and priority-based scheduling. Typically these approaches use a heuristic that gives a static [34, 43] or biased [42] partition of the CPU to either real-time threads or non-real-time threads. A new approach is taken by the SMaRT scheduler, which dynamically balances between the needs of interactive, batch and real-time processes by giving proportional shares to each process and a bias that improves the responsiveness of interactive and real-time threads [82].

The BERT scheduler [14] handles both real-time and non-real-time threads using the same scheduling mechanism. Processes submit units of work to be scheduled, and the scheduler creates deadlines for the work based on previous measures of the work's time to completion. BERT automatically assesses whether a given thread will meet its deadline, and if not can either steal cycles from a lower priority thread or can cancel the thread. BERT is similar in philosophy to our feedback-scheduling approach since it uses feedback of past execution times in its scheduling, but it does not use or measure application progress and as such is subject to the same problems as traditional schedulers.

7.4.4 Feedback-based Scheduling

Software feedback has been used extensively to build adaptive operating system schedulers. In Unix operating systems, multi-level feedback queue scheduling [22] is used to schedule processes. The system monitors each thread to see whether it consumes its entire time slice or does I/O and adjusts its priority accordingly. An I/O bound process gets higher priority so that the latency of interactive processes does not suffer due to CPU bound processes. Feedback-based resource management is also used for memory management in Windows NT [24].

Our feedback work is heavily influenced by Massalin and Pu's work who pioneered the idea of using feedback control for fine-grained resource management in operating systems [71] and used it in the design of the Synthesis kernel [70]. Synthesis used run-time code synthesis on frequently used kernel routines to create specialized versions of executable code at run-time which allowed fast interrupt processing, context switching and thread dispatch with low overhead. The feedback-based fine-grain scheduling mechanism performed frequent scheduling actions and policy adjustments, resulting in an adaptive and self-tuning system.

Synthesis also used fine-grained adaptive scheduling of interdependent jobs such as threads in a pipeline. Each pipeline consisted of multiple threads that were coupled by their input and output queues. Data elements such as audio samples were passed along the pipeline stage-by-stage, and processed at each stage and the length of a thread's input/output queue was a measure of the thread's progress in the pipeline. The main problem with this control approach in Synthesis is that the controller behavior at each thread is dependent on the controller behavior at other threads. For example, a poor controller at one stage can destabilize the output of other controllers that are

otherwise well constructed because the input and output fill levels are correlated. Our solution to this problem is to use time-stamps on data packets, which allows each controller to estimate a thread's progress independent of the behavior of other controllers in the pipeline. With this approach, the behavior of each controller can be analyzed independently.

The second problem with the Synthesis kernel is that the kernel is highly specialized and its feedback approach could not be easily extended to support low-latency applications in general-purpose systems. This thesis shows that the ideas behind Synthesis can also be applied to standard general-purpose operating systems.

We had initially implemented a feedback controller using some of the same techniques as Synthesis [104]. In particular, this controller also used the fill level of the input and output queues to measure progress and hence suffered from the same problem of the controller behavior at each thread being dependent on the behavior of all other dependent threads. There were two other problems with our initial approach. First, the end points of a pipeline had to be externally driven using a different scheduler. If all threads are scheduled using the feedback controller, then the input and output of processes can be matched without necessarily providing the correct rate to the pipeline! This problem arises because the scheduler has not explicit knowledge of the timing information of the thread. Our current real-rate scheduler solves this problem by explicitly requiring timing information (but not the resource requirements) from threads. This approach allows any thread, including all threads in a pipeline to be scheduled by our scheduler.

The second problem with our initial approach was that we used a simple linear PID controller where the proportional gain of the thread was not estimated directly. Instead, a fixed gain parameter was used to drive the control law. This approach caused problems when applications had very different requirements. The controller could be tuned for a given proportion such as 40%. However, the same controller was either too responsive or too slow in responding if the proportion requirements of a thread were 4% or 80%. Our current controller solves this problem with explicitly estimating the proportional gain of the thread.

In addition, our initial controller had several parameter that were tuned using ad-hoc techniques (a.k.a magic)! Our current controller has two parameters α and β. Our simulation experiments have shown that the real-rate controller performs optimally under a wide range of conditions when α is fixed at 1/8 and β is fixed at 1/2. Hence, no further parameter tuning is needed! The

only additional requirement of this controller is that applications specify their progress needs in terms of their timing requirements.

Our feedback scheduling solution is similar to rate-based scheduling proposed by Jeffay and Bennett [56], in that resources are allocated based on rate specifications of x units of execution every y time units. However, their units are events which are converted to CPU cycles using a worst-case estimate of event processing time. Applications must specify x, y, and the worst-case estimation, and an upper-bound on response time. In addition, these values are constant for the duration of the application. Their system also uses pipelines of processes so that dependent stages do not need to specify their rate, merely their event processing time. In contrast, our system provides dynamic estimation and adjustment of rate parameters, and only requires that the progress metric be specified.

Stankovic [103] uses a feedback controlled earliest-deadline first (FC-EDF) algorithm to adjust allocations of threads in order to reduce the thread's missed deadlines. While this approach uses missed deadlines as an input to the feedback controller, our feedback approach uses timestamps and does not need missed deadlines to monitor progress. The use of feedback scheduling allows control-based analysis of the scheduler performance. Abeni [4] uses such analysis to prove the stability of their scheduling mechanism.

TCP [53] is another adaptive resource allocator that uses buffers to estimate an application's resource requirements. It uses the fill level in its send and receive buffers to allocate network bandwidth to each flow. For example, if the send buffer is periodically empty or the receiver buffer is full, then TCP assumes that the flow is application limited and reduces its sending rate.

7.5 Visualization of Low-Latency Applications

We have implemented a user-level software visualization tool and library called gscope that helps in the visualization and debugging of low-latency applications and borrows some of its ideas from an oscilloscope. The oscilloscope is essentially a graph-displaying device – it draws a graph of an electrical signal. Oscilloscopes can help determine various signal properties: time and voltage values of a signal, frequency of an oscillating signal, phase difference between two oscillating signals, a malfunctioning component that is distorting the signal, AC and DC components of a

signal and noise in a signal. Oscilloscopes can process analog or digital waveforms and store data for later viewing and printing.

The ideas in gscope were heavily influenced by a software oscilloscope that was designed by Cen in our group for visualizing the behavior of software feedback circuits [17]. Gscope is similar in functionality to `gstripchart` [62], a stripchart program that charts various user-specified parameters as a function of time such as CPU load and network traffic levels. The gstripchart program periodically reads data from a file, extracts a value and displays these values. However, unlike Gscope, `gstripchart` has a configuration file based interface rather than a programmatic interface, which limits its use for debugging or modifying system behavior.

There is a large body of work related to implementing software digital oscilloscope functionality for audio visualization. The basic idea is to record sound with a microphone and then display the digitized sound waves. `Xoscope` [112] is one such program. `Xmms` [7] displays sound frequency during audio playback. `Baudline` [13] is a real-time signal analysis tool and an offline time-frequency browser. These programs emulate the functionality of a digital oscilloscope much more closely than gscope. However, these programs are focusing on audio visualization while gscope focuses on visualization and debugging of software behavior. Thus certain oscilloscope features are not appropriate for gscope and vice-versa.

There are hundreds of measurement tools that can be used for capturing system and network performance [84]. Gscope complements them because it can be used to visualize their output in real-time.

Chapter 8

Conclusions and Future Work

In the past, general-purpose OSs have tended to focus on the performance of throughput-oriented applications, such as compilation jobs, file transfers and database operations and thus the design and structure of OSs has emphasized throughput over all other metrics. Unfortunately, this design is insufficient and inappropriate for low-latency applications that have tight timing requirements because current OSs, in an attempt to improve system throughput, sacrifice control over the precise times at which applications can be scheduled. For example, a general-purpose OS such as Linux provides coarse-grained 10 ms timing resolution to applications and has non-preemptible sections that can be as large as 20-100 ms under heavy load. The timing needs of low-latency applications cannot be satisfied under these conditions. For example, soft modems require execution every 12.5 ms or else the modem throughput goes down significantly and eventually the modem loses connection.

The benefit of using general-purpose operating systems for low-latency applications is lower cost and flexibility. It is cheaper to use commodity hardware and software than dedicated hardware or specialized operating systems for such applications. Using software instead of hardware allows more extensibility and flexibility and reduces the time to market new products. In addition, when source code is available, users can help fix software bugs thus potentially reducing the cost of maintaining these applications. Thus, in the future, we expect to see the software development of software radio [15, 20], an active area of research activity. These radios use complex adaptive modulation techniques that are only possible with software implementations. Jones [60] reports that software implementations of the 802.11b wireless LAN protocol and the Bluetooth wireless protocol are possible, and while they only require 2-3% of a 600 MHz CPU, they require short computations extremely frequently, every 312.5 μs. To effectively support these new applications

in general-purpose OSs requires a new vision that encourages the design of systems that provide good support for traditional as well as low-latency applications and allows tuning the system for both classes of applications.

A different strategy is employed by real-time systems that provide very fine-grained timing control to threads. However, these systems are highly specialized and tend to ignore the performance of throughput-oriented applications. In addition, they require very precise specification of the timing requirements of applications that may be hard to determine statically.

The motivation for this work stems from the observation that there is a large class of applications that have an intermediate level of timing control requirements. These "low-latency" applications are neither as loose as throughput-oriented applications, where the timing of individual operations in the application is not important, in their timing requirements, nor as tight as hard real-time applications such as nuclear-reactor or spaceship control where errors can be disastrous. Some researchers prefer to use the term "soft real-time" for such applications. However, we believe that the term "low-latency" is more appropriate because these applications generate real-world events and require low latency between the arrival of an event and its response. The OS provides good support for such applications when it introduces low latency in the processing of events. Further, low-latency applications such as video conferencing may not have real deadlines. For example, the 50 ms delay requirement for a conferencing application at the application and OS level is a suggested delay and by no means a requirement. This application really requires low-latency at the application and the OS level for high quality conferencing and imposing deadlines serves to only limit scheduling flexibility. The spectrum of low-latency applications between hard real-time and throughput-oriented applications becomes even more fuzzy with quality-adaptive applications, which can reduce their processing needs when their bottleneck resource is overloaded. In this case, the notion of deadlines is even more vague and the best the OS can do is to provide a tunable low-latency, high throughput system because the final quality realized in these applications depends on a combination of these metrics. For example, in a video conferencing application, the overall quality may depend on the total number of packets that arrive within a given delay. This metric has both throughput and latency metrics embedded in it.

The observation that applications have a range of throughput and latency requirements motivated the design of Time-Sensitive Linux (TSL). The goal of TSL is to support a range of low-latency applications without significantly affecting the performance of other throughput-oriented applications running on the system. We started by identifying sources of latency in general-purpose operating systems and showed that the timing mechanism, non-preemptible kernel sections, scheduling and output buffering are the main sources. We evaluated the behavior of a general-purpose OS such as Linux and measured the latency caused by each of these sources. The timing mechanism is coarse and has a 10 ms timer latency. Non-preemptible sections in standard Linux are large and can cause 20-100 ms preemption latency. We measured output buffering latency for a specific class of network applications that use TCP for streaming data and showed that this latency can be as large as 2-4 s even though the network round-trip time is only 100 ms.

Scheduling latency is unique because it depends on the accuracy of the implementation and the application's choice of the specific scheduling mechanism. The accuracy of the scheduler implementation, interestingly, depends on timer and non-preemption latency because a scheduler is itself a low-latency application. The choice of the scheduling mechanism depends on how easy it is for the application to choose to use the appropriate mechanism. Our evaluation showed that a low-latency application such as the `mplayer` multimedia player that uses normal Linux priorities can experience scheduling latencies as large as 60 ms when run together with just one additional CPU intensive process.

After evaluating each source of latency in a commodity OS, we proposed techniques that help to reduce these latency sources. An important goal of each of these techniques is to minimize their effect on system throughput. For timer latency, we proposed, implemented and evaluated firm timers that provide high resolution timing with low overhead both at the kernel and the user level. Firm timers combine the benefits of one-shot timers with periodic timers and soft timers. The use of one-shot timers helps provide timing resolution that is close to interrupt service times. The use of periodic timers for long timeouts allows using more efficient data structures and soft timers reduce the number of timer interrupts that need to be generated for high resolution timing. Our measurements show that the maximum firm timer overhead on TSL is less than 1.5% as compared to standard Linux. In addition, the use of soft timers and an overshoot parameter allows making a trade-off between timer latency and system overhead.

Next, we described the basic approaches that have been proposed for reducing preemption latency. The first approach is explicit insertion of preemption points at strategic points inside the kernel so that a thread yields the CPU after it has executed for some time in the kernel. In this way, the size of non-preemptible sections is reduced. This approach requires careful auditing of the kernel under heavy system load to determine long kernel paths. In addition, the placement of preemption points can be a time-intensive process. The second approach is to allow preemption anytime the kernel is not accessing shared kernel data. To support this fine level of kernel preemptibility, all shared kernel data must be explicitly protected using mutexes or spinlocks. Then preemption is always allowed except when a spinlock is held or an interrupt handler is executing. This approach can have high preemption latency when spinlocks are held for a long time. The third approach combines the first two approaches by putting explicit preemption points wherever spinlocks are held for a long time.

We evaluated several variants of the Linux kernel that implement these different preemption strategies under heavy system load. Our evaluation showed, a little surprisingly, that the explicit preemption approach performs better than the preemptible kernel approach. One reason may be that the numerous preemption possibilities in a preemptible kernel lead to long kernel paths that are not exercised under normal conditions. However, the combination of the two approaches has the lowest preemption latency under most situations. This result shows that auditing of kernel code for long kernel paths is an essential requirement for achieving low preemption latency.

To reduce output-buffering latency, we proposed an adaptive buffer sizing technique for TCP flows. This latency can be a significant portion of end-to-end latency in such flows. Hence, reducing output-buffering latency allows streaming with very low end-to-end latency.

Adaptive buffer sizing separates the two functions performed by TCP's send buffer. In standard TCP, the send buffer keeps packets for retransmission and performs output buffering to improve network throughput. For low-latency network streaming, the send buffer should be used mainly for keeping packets that may need to be retransmitted since these packets do not add any additional buffering delay unless they are actually retransmitted. This approach reduces output latency but it also reduces network throughput. We analyze the reasons for this loss in network throughput and then propose keeping a few extra packets (in addition to the packets that are currently being transmitted) to regain much of the lost network throughput. Our evaluation shows

that keeping only three extra packets allows achieving 90-95% TCP throughput while maintaining very low output-buffering latency.

We have integrated the techniques described above into an extended version of Linux that we call Time-Sensitive Linux (TSL). TSL is a responsive kernel with an accurate timing mechanism and it allows network streaming over TCP with low latency. We evaluated TSL and showed that it improves the performance of real low-latency applications significantly. We used a non-streaming video application, `mplayer`, and showed that TSL helps to improve its audio/video synchronization under different types of user and system loads. Then we used a streaming low-latency media application and showed that TSL helps to improve the end-to-end latency experienced by packets in this application.

A key motivation for TSL is to provide an infrastructure for accurately implementing real-time CPU scheduling algorithms and thus enable applying scheduling analysis and real-time guarantees to a general-purpose system. Traditionally, real-time scheduling mechanisms such as priority-based scheduling and proportion-period scheduling have been used to provide predictable and low scheduling latency. Unfortunately, the problem with priority-based scheduling is that it lacks sufficient control over allocating resources. For example, it is hard to correlate the CPU requirements of an application with a specific priority assignment. In addition, priority-based scheduling can lead to starvation and priority-inversion, both of which reduce system robustness.

The second class of real-time scheduling algorithms is reservation-based algorithms, such as proportion-period scheduling, that can provide fine-grained control over resource allocation (such as a percentage of CPU every period of time). These schedulers can avoid starvation by assigning a minimum amount of CPU to each thread. However, these schedulers have not found general applicability because of the difficulty associated with specifying the resource requirements of applications, such as proportion of CPU every period. In a general-purpose system, an application's resource requirements are often not known or can be hard to determine statically. For example, the soft modem application requires different proportions of CPU on different CPUs.

In this thesis we have described a new approach to scheduling called real-rate scheduling that assigns proportion based on the measured rate of progress. Our system utilizes progress monitors such as time-stamps on packet data, a feedback-based controller that dynamically adjusts the CPU allocation and period of threads in the system, and an underlying proportional-period

scheduler. As a result, our system dynamically adapts allocation to meet current resource needs of applications, without requiring input from human experts.

The accuracy of allocating resources using our feedback-based real-rate controller depends (among other factors) on the accuracy of actuating proportions and periods in a proportion-period scheduler. For example, inaccuracy in policing proportions introduces 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 evaluation shows that the underlying proportion-period scheduler, which is itself a low-latency application, can be implemented much more accurately under TSL as compared to standard Linux.

The real-rate scheduler uses time-stamps to monitor application progress. We showed how a real-rate system can be modeled and a controller designed for this non-linear system using standard feedback linearization techniques. The goal of the controller is to limit proportion overshoot and minimize thread delay. Our analysis showed that delay can be minimized by taking the granularity of time-stamps into account and using a sampling period that is 3 to 5 times the time-stamp granularity. This approach allowed us to tune the control parameters for optimal delay in a systematic and very generic way because it made very few assumptions about the operating environment.

Finally, we described *gscope*, a tool designed for visualizing low-latency software applications. Its goal is to reduce the cycle time needed for testing and debugging such applications by providing an oscilloscope-like interface that can be integrated with the application. We have used gscope successfully in many of our applications and have built several compelling demonstrations of our research work using this tool.

Future Work

This section discusses new research problems that emerge from this dissertation and potential solutions to these problems. In this thesis, we have explored the design and implementation of a general-purpose operating system that can support applications requiring low-latency response and fine-grained scheduling. We have implemented our approach on Time-Sensitive Linux (TSL) and have shown that it has low overhead on throughput-oriented applications and thus it can be used effectively for low-latency as well as general-purpose applications.

General-purpose OSs have traditionally been benchmarked for throughput. We have developed a latency benchmark suite that measures the different sources of latency under heavy system load. However, a more comprehensive latency benchmark suite is needed that stresses every part of the system. Ideally, as the system is extended, benchmarks should be developed that stress the extensions, so that system latency can be reevaluated. In addition, this benchmark for measuring latency should be integrated with tools such as gscope and LTT [113] for visualizing latency behavior.

Although our initial results are promising, TSL still needs further investigation, since there are open issues related to firm timer performance and kernel preemptibility under heavy interrupt loads. For firm timers, we are interested in investigating whether real workloads commonly lead to the $S/C > K$ condition (see Section 2.4.2) under which firm timers have lower overhead than hard timers. For kernel preemptibility, we plan to extend our evaluation of preemption latency to separate preemption latency caused by the kernel (such as when acquiring/releasing spinlocks or disabling/enabling interrupts) from interrupt processing latency. For example, in Linux, an intensive interrupt load can cause long latencies due to the design of the interrupt processing mechanism (ISRs, tasklets, and bottom halves). One way to improve the performance of low-latency applications in the presence of heavy interrupt load is to schedule them as real-time threads and defer certain parts of interrupt processing until after real-time threads. Another approach that mitigates the effects of interrupt processing is using resource reservations together with some adaptation strategy [93, 3, 1].

The TCP-based streaming experiments showed that MIN_BUF TCP streams have low end-to-end latency. These results are based on experiments conducted over an experimental network test-bed with a simple topology where all flows had the same round-trip time. It would be useful to conduct these experiments with a more complex topology where different flows have different round-trip times since TCP behavior is strongly dependent on round-trip times.

We have explored adapting the send buffer using three different sizes for MIN_BUF(A, B) flows. These different configurations, with increasing buffer sizes, have increasing latency and throughput. Another approach for adapting the send buffer is to auto-tune the values of A and B so that the send buffer contributes a certain amount of delay while providing the best possible throughput.

For low latency streaming, packets should have low protocol latency but also low variation in inter-arrival times (low jitter). For example, a quality-adaptive application sees bursty arrivals as either bursty quality, or if it buffers the bursts then as higher end-to-end latency. Packet inter-arrival times depend on packet sending times and protocol latency. A MIN_BUF TCP flow reduces protocol latency but does not directly attempt to change the time at which packets are transmitted by TCP. In TCP, the packet sending time is determined by the arrival of acknowledgments which open the window so more packets can be sent. If acknowledgments arrive regularly, then packets can be sent regularly. However, if acknowledgment arrivals are bursty [102], TCP sends packets in bursts. The solution for improving TCP's bursty packet transmission (and thus inter-arrival) behavior is to either pace the application sending rate or pace the TCP sending rate [8, 66, 5, 73].

An important design choice for TSL is providing low-latency support with low overhead for throughput-oriented applications. Eventually, we believe that OSs should be designed so that they can be tuned either automatically or manually for both classes of applications. For example, firm timers allow tuning for timer latency versus system overhead by exposing an overshoot parameter. Similarly, MIN_BUF TCP streams have parameters for tuning the trade-off between latency and throughput. It should also be possible to tune preemption latency based on application requirements. For example, the number of iterations a long-running non-preemptible function executes before it invokes the scheduler could be tuned automatically based on past execution times of the function.

Our feedback scheduling approach deals with the problem of determining the resource requirements of low-latency applications so that a robust and predictable scheduling scheme like reservation-based scheduling can be used to schedule these applications in a general-purpose environment. Our scheduler automatically converts an application-specific metric such as application progress to resource requirements. This approach, while promising, raises several possibilities for future work. Our experience with feedback scheduling is limited to simulated applications and a video playback application. We would like to analyze our feedback scheduler with other low-latency applications that have tighter latency requirements such as soft modems.

The real-rate implementation uses time-stamps to determine progress. Fine-grained time-stamps, such as time-stamps for smaller size packets, improve the responsiveness of the controller but they also add space and processing overhead. We would like to characterize this overhead and

study the trade-off between this overhead and the effectiveness of the control response.

Bibliography

[1] Luca Abeni. Coping with interrupt execution time in real-time kernels: a non-intrusive approach. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 1–5, December 2001. Work-In-Progress.

[2] 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 *Proceedings of the IEEE Real Time Technology and Applications Symposium*, pages 133–143, September 2002.

[3] Luca Abeni and Giuseppe Lipari. Compensating for interrupt process times in real-time multimedia systems. In *Third Real-Time Linux Workshop*, November 2001.

[4] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 71–80, December 2002.

[5] Amit Aggarwal, Stefan Savage, and Thomas Anderson. Understanding the performance of TCP pacing. In *Proceedings of the IEEE Infocom*, pages 1157–1165, 2000.

[6] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. Internet RFC 2581, April 1999.

[7] Peter Alm, Thomas Nilsson, and et al. XMMS: A cross platform multimedia player. `http://www.xmms.org`, viewed in Jan 2002.

[8] Mohit Aron and Peter Druschel. TCP: Improving startup dynamics by adaptive timers and congestion control. Technical Report TR98-318, Rice University Computer Science, 1998.

[9] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, August 2000.

[10] P. Bagal, S. Kalyanaraman, and B. Packer. Comparative study of RED, ECN and TCP rate control. Dept of ECSE, RPI, 1999.

[11] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for internet hosts. In *Proceedings of the ACM SIGCOMM*, pages 175–187, 1999.

[12] Michael Barabanov and Victor Yodaiken. Real-time Linux. *Linux Journal*, 34, February 1997.

[13] The Baudline real-time signal analysis tool. `http://www.baudline.com`, viewed in Jan 2003.

[14] A. Bavier, L. Peterson, , and D. Moseberger. Bert: A scheduler for best effort and realtime tasks. Technical Report TR-587-98, Princeton University, August 1998.

[15] Vanu Bose, Mike Ismert, Matt Welborn, and John Guttag. Virtual radios. *IEEE Journal of Selected Areas in Communication*, 17(4):591–602, April 1999.

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

[17] Shanwei Cen. *A Software Feedback Toolkit and Its Applications in Adaptive Multimedia Systems*. PhD thesis, Oregon Graduate Institute of Science and Technology, August 1997. Department of Computer Science and Technology.

[18] Edward Y. Chang and Hector Garcia-Molina. BubbleUp: Low latency fast-scan for media servers. In *Proceedings of the ACM Multimedia*, pages 87–98, 1997.

[19] Wu chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang S. Shin. Techniques for eliminating packet loss in congested TCP/IP networks. Technical Report CSE-TR-349-97, U. Michigan, November 1997.

[20] John Chapin and Vanu Bose. The vanu software radio system. `http://www.vanu.com/publications/vanuinc-sdrforum-sd-submitted.pdf`, viewed in Jun 2003.

[21] Kimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Traffic characteristics of the T1 NSFNET backbone. In *Proceedings of the IEEE Infocom*, pages 885–892, 1993.

[22] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 335–344, 1962.

[23] Intel Corporation and Microsoft Corporation. *PC 99 System Design Guide - A Technical Reference for Designing PCs and Peripherals for the Microsoft Windows Family of Operating Systems*, 1998. Chapter 19 - Modems. `ftp://download.intel.com/design/pc98/pc99/Pc_99_1.pdf`, viewed in Sep 2002.

[24] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[25] Miguel de Icaza and et al. The Gnome desktop environment. `http://www.gnome.org`, viewed in Jan 2002.

[26] Jayanta K. Dey, Subhabrata Sen, James F. Kurose, Donald F. Towsley, and James D. Salehi. Playback restart in interactive streaming video applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 458–465, 1997.

[27] Constantinos Dovrolis, Dimitrios Stiliadis, and Parameswaran Ramanathan. Proportional differentiated services: Delay differentiation and packet scheduling. In *Proceedings of the ACM SIGCOMM*, pages 109–120, 1999.

[28] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, October 1996.

[29] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2000.

[30] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *ACM Computer Communication Review*, 26(3):5–21, July 1996.

[31] Wu-Chi Feng, Ming Liu, Brijesh Krishnaswami, and Arvind Prabhudev. A priority-based technique for the best-effort delivery of stored video. In *Proceedings of the SPIE Multimedia Computing and Networking Conference*, pages 286–300, January 1999.

[32] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *ACM/IEEE Transactions on Networking*, 1(4):397–413, August 1993.

[33] Sally Floyd, Mark Handley, and Eddie Kohler. Problem statement for DCP. Work in progress, IETF Internet Draft draft-floyd-dcp-problem-00.txt, expires Aug 2002, Feb 2002.

[34] Bryan Ford and Sai Susarla. Cpu inheritance scheduling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 91–105, 1996.

[35] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Addison-Wesley, third edition, 1994.

[36] Gene F. Franklin, J. David Powell, and Michael Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, third edition, 1997.

[37] M. Gaynor. Proactive packet dropping methods for TCP gateways. `http://www.eecs.harvard.edu/~gaynor/final.ps`, viewed in Dec 2001, October 1996.

[38] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on a commodity OS. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 165–180, December 2002.

[39] Ashvin Goel, Charles Krasic, Kang Li, and Jonathan Walpole. Supporting low latency TCP-based media streams. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*, pages 193–203, May 2002.

[40] Ashvin Goel, Molly H. Shor, Jonathan Walpole, David C. Steere, and Calton Pu. Using feedback control for a network and CPU resource management application. In *Proceedings of the 2001 American Control Conference (ACC)*, June 2001. Invited paper.

[41] Ashvin Goel and Jonathan Walpole. Gscope: A visualization tool for time-sensitive software. In *Proceedings of the Freenix Track of USENIX Technical Conference*, pages 133–142, June 2002.

[42] Ramesh Govindan and David P. Anderson. Scheduling and ipc mechanisms for continuous media. In *Proceedings of the Symposium on Operating Systems Principles*, pages 68–80, October 1992.

[43] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 107–121, October 1996.

[44] The GTK graphical user interface toolkit. `http://www.gtk.org`, viewed in Jun 2002.

[45] Hao hua Chu and Klara Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 296–301, June 1999.

[46] Jie Huang, Charles Krasic, Jonathan Walpole, and Wu chi Feng. Adaptive live video streaming by priority drop. In *Proceedings of the IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, July 2003. To appear.

[47] Bradley Huffaker, Marina Fomenkov, David Moore, and kc claffy. Macroscopic analyses of the infrastructure: Measurement and visualization of internet connectivity and performance. In *A workshop on Passive and Active Measurements (PAM2001)*, April 2001.

[48] P. Hurley and J. Y. Le Boudec. A proposal for an asymmetric best-effort service. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*, pages 132–134, May 1999.

[49] Gianluca Iannaccone, Martin May, and Christophe Diot. Aggregate traffic performance with active queue management and drop from tail. *ACM Computer Communication Review*, 31(3):4–13, July 2001.

[50] Microsoft Inc. Windows Netmeeting. `http://www.microsoft.com/netmeeting`, viewed in Jun 2002.

[51] International Telecommunication Union (ITU). *Transmission Systems and Media, General Recommendation on the Transmission Quality for an Entire International Telephone Connection; One-Way Transmission Time*. Geneva, Switzerland, March 1993. Recommendation G.114, Telecommunication Standardization Sector of ITU.

[52] Ixia. Ixia 1600 Network Traffic Generator. `http://www.ixiacom.com/`, viewed in Jul 2002.

[53] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM*, pages 314–329, August 1988.

[54] V. Jacobson, K. Nichols, and K. Poduri. An expedited forwarding PHB. Internet RFC 2598, June 1999.

[55] Ramesh Jain. Teleexperience: Communicating compelling experiences. Keynote speech at ACM Multimedia 2001.

[56] Kevin Jeffay and David Bennett. A rate-based execution abstraction for multimedia computing. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 67–77, April 1995.

[57] E. Johnson and A. Kunze. *IXP1200 Programming*. Intel Press, March 2002.

[58] M. B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the Symposium on Operating Systems Principles*, pages 198–211, October 1997.

[59] 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 *Proceedings of the ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[60] Michael B. Jones and Stefan Saroiu. Predictability requirements of a soft modem. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–49, June 2001.

[61] Rudolph E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[62] John Kodis. Gstripchart: A stripchart-like plotting program. `http://users.jagunet.com/~kodis/gstripchart/gstripchart.html`, viewed in Mar 2002.

[63] Charles Krasic. The Mxtraf traffic generator. `http://www.sf.net/projects/mxtraf`, viewed in Sep 2002.

[64] Charles Krasic, Kang Li, and Jonathan Walpole. The case for streaming multimedia with TCP. In *Proceedings of the International Workshop on Interactive Distributed Multimedia Systems (iDMS)*, pages 213–218, September 2001.

[65] Charles Krasic, Jonathan Walpole, and Wu chi Feng. Quality-adaptive media streaming by priority drop. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 112–121, June 2003.

[66] Joanna Kulik, Robert Coulter, Dennis Rockwell, and Craig Partridge. A simulation study of paced TCP. Technical Report BBN-TM-1218, BBN Technologies, August 1999.

[67] 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.

[68] Robert Love. The Linux kernel preemption project. `http://kpreempt.sf.net`, viewed in Mar 2001.

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

[70] Henry Massalin and Calton Pu. Input/output in the synthesis kernel. In *Proceedings of the Symposium on Operating Systems Principles*, pages 191–200, December 1989.

[71] Henry Massalin and Calton Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, Winter 1990. Special Issue on selected papers from the Workshop on Experiences in Building Distributed Systems, October 1989.

[72] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. Internet RFC 2018, October 1996.

[73] Matt Mathis, Jeff Semke, Jamshid Mahdavi, and Kevin Lahey. Rate-halving algorithm for TCP congestion control, June 1999. Work in Progress, Internet Draft `http://www.psc.edu/networking/ftp/papers/draft-ratehalving.txt`, viewed in Feb 2002.

[74] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgment: Refining tcp congestion control. In *Proceedings of the ACM SIGCOMM*, pages 281–291, October 1996.

[75] 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*, pages 90–99, May 1994.

[76] Clifford W. Mercer and Hideyuki Tokuda. Preemptibility in real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 78–88, December 1992.

[77] D. L. Mills. Network time protocol (version 3) specification, implementation and analysis. Technical report, University of Delaware, March 1992. DARPA Network Working Group Report RFC-1305.

[78] Montavista Software - Powering the embedded revolution. `http://www.mvista.com`, viewed in May 2002.

[79] Andrew Morton. Linux scheduling latency. `http://www.zip.com.au/~akpm/linux/schedlat.html`, viewed in Sep 2001.

[80] Mplayer - Movie player for Linux. `http://www.mplayerhq.hu`, viewed in Jun 2002.

[81] T. Nandagopal, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. Relative delay differentation and delay class adaptation in core-stateless networks. In *Proceedings of the IEEE Infocom*, pages 421–430, March 2000.

[82] Jason Nieh and Monica Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the Symposium on Operating Systems Principles*, pages 184–197, October 1997.

[83] NIST. The NIST network emulation tool. `http://www.antd.nist.gov/itg/nistnet`, viewed in Jun 2002.

[84] NLANR network performance and measurement tools. `http://dast.nlanr.net/npmt`, viewed in Nov 2001.

[85] Brian Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin Walker. Agile application-aware adaptation for mobility. In *Proceedings of the Symposium on Operating Systems Principles*, pages 276–287, October 1997.

[86] Jörg Nonnenmacher, Ernst W. Biersack, and Don Towsley. Parity-based loss recovery for reliable multicast transmission. *ACM/IEEE Transactions on Networking*, 6(4):349–361, 1998.

[87] Shuichi Oikawa and Raj Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 111–120, December 1998. Work-In-Progress Session.

[88] K. Pentikosis, H. Badr, and B. Kharmah. TCP with ECN: Performance gains for large transfers. Technical Report SBCS-TR-2001/01, Department of Computer Science, SUNY Stony Brook, March 2001.

[89] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the USENIX Technical Conference*, pages 213–224, 1995.

[90] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. Internet RFC 3168, September 2001.

[91] N. Reddy. Improving latency in interactive video server. In *Proceedings of the SPIE Multimedia Computing and Networking Conference*, pages 108–112, February 1997.

[92] John Regehr. Inferring scheduling behavior with Hourglass. In *Proceedings of the Freenix Track of the 2002 USENIX Annual Technical Conference*, pages 143–156, June 2002.

[93] John Regehr and John A. Stankovic. Augmented CPU reservations: Towards predictable execution on general-purpose operating systems. In *Proceedings of the IEEE Real Time Technology and Applications Symposium*, pages 141–148, May 2001.

[94] Reza Rejaie, Mark Handley, and Deborah Estrin. Quality adaptation for congestion controlled video playback over the internet. In *Proceedings of the ACM SIGCOMM*, pages 189–200, 1999.

[95] Reza Rejaie, Mark Handley, and Deborah Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *Proceedings of the IEEE Infocom*, pages 1337–1345, 1999.

[96] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, April 1997.

[97] J. Hadi Salim and U. Almed. Performance evaluation of explicit congestion notification (ECN) in IP networks. Internet RFC 2884, July 2000.

[98] S. Savage and H. Tokuda. RT-Mach timers: Exporting time to the user. In *Proceedings of the USENIX Mach Symposium*, pages 111–118, April 1993.

[99] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic TCP buffer tuning. In *Proceedings of the ACM SIGCOMM*, pages 315–323, 1998.

[100] Benno Senoner. Scheduling latency tests / high performance low latency audio. `http://www.gardena.net/benno/linux/audio`, viewed in Jun 2002.

[101] 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.

[102] Scott Shenker, Lixia Zhang, and David Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proceedings of the ACM SIGCOMM*, pages 133–147, 1991.

[103] John A. Stankovic, Chenyang Lu, Sang H. Son, and Gang Tao. The case for feedback control real-time scheduling. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 11–20, June 1999.

[104] 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 USENIX Symposium on Operating Systems Design and Implementation*, pages 145–158, February 1999.

[105] David Steere, Molly H. Shor, Ashvin Goel, Jonathan Walpole, and Calton Pu. Control and modeling issues in computer operating systems: Resource management for real-rate computer applications. In *Proceedings of the IEEE Conference on Decision and Control (CDC)*, December 2000.

[106] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. Internet RFC 2960, Oct 2000.

[107] I. Stoica, H. Abdel-Wahab, , and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. In *Proceedings of the SPIE Multimedia Computing and Networking Conference*, pages 207–214, February 1997.

[108] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 288–299, December 1996.

[109] R. van der Merwe and E. A. Wan. The square-root unscented Kalman filter for state and parameter-estimation. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 3461 –3464, May 2001.

[110] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 1–12, November 1994.

[111] Clark Williams. Linux scheduler latency. `http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf`, viewed in Mar 2002.

[112] Timothy D. Witham. Xoscope: A digital oscilloscope for Linux. `http://xoscope.sf.net`, viewed in Mar 2002.

[113] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the USENIX Technical Conference*, pages 13–26, June 2000.

[114] David K. Y. Yau and Siman S. Lam. Adaptive rate controlled scheduling for multimedia applications. *ACM/IEEE Transactions on Networking*, 5(4):475–488, August 1997.

[115] 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*, pages 11–20, October 1998.

Appendix A

Feedback Linearization

The goal of feedback linearization is to design a control law for a non-linear system model so that the closed loop dynamics, or the dynamics of the system together with the controller, are linear. Here we describe the basic feedback linearization technique for a single input, single output system. Given a system model, $x_{k+1} = x_k + g_k u_k$, where u_k is the input to the system at time k, x_k is the output of the system that can be measured and g_k is the non-linearity in the system, the control law that produces linear closed-loop error dynamics is given by the following equation. Note that x_k is the input while u_k is the output of the control law.

$$u_k = (g_k)^{-1}[x_{k+1}^{des} - x_k - \tau(x_k - x_k^{des})] \tag{A.1}$$

In Equation A.1, $(g_k)^{-1}$ inverts the non-linearity in the system model. The superscript `des` describes the desired values of x_k and x_{k+1}, and τ is called the gain of the controller. Substituting the control law in the system model gives us the control-equation as $x_{k+1} = x_{k+1}^{des} - \tau(x_k - x_k^{des})$. Or rearranging the terms,

$$x_{k+1} - x_{k+1}^{des} + \tau(x_k - x_k^{des}) = 0 \tag{A.2}$$

If $x_k - x_k^{des}$ is defined as the error e_k in the system (it is the difference in the output of the system from the desired value), then Equation A.2 is simply $e_{k+1} + \tau e_k = 0$. Using basic control theory, we know that this closed loop control is stable when the absolute values of it eigenvalues λ are less than one. The eigenvalue equation of this simple closed-loop equation is $\lambda + \tau = 0$. Hence, when the gain $|\tau| < 1$, the control law will be stable. This linearization process assumes that the non-linearity g_k can be precisely inverted. If there is an error in the inversion, then the

gain τ must be chosen carefully within the $|\tau| < 1$ requirements, based on understanding and analyzing the source of the error.