# THE DESIGN, IMPLEMENTATION, AND EVALUATION OF SMART: A SCHEDULER FOR MULTIMEDIA APPLICATIONS

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

**Jason Nieh**

**June 1999**

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Monica S. Lam (Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John L. Hennessy

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

S. Simon Wong

Approved for the University Committee on Graduate Studies:

# Abstract

Multimedia applications are becoming ubiquitous. Unlike conventional interactive and batch applications, these applications often have real-time requirements. As multimedia applications are integrated with conventional non-real-time applications in the general-purpose computing environment, the problem arises of how to support the resulting mix of activities. A key question is how does the operating system schedule processor cycles to enable applications with and without real-time requirements to co-exist and run effectively?

To address this question, we have created SMART, a Scheduler for Multimedia And Real-Time applications. SMART explicitly supports the time constraints of real-time applications, and provides dynamic feedback to these applications to allow them to adapt their performance based on the availability of processor cycles. It is unique in its ability to make efficient use of processor cycles in meeting real-time requirements under a dynamically varying system load, even in the absence of admission control policies when the system is overloaded. SMART integrates support for real-time and non-real-time applications. This allows it to provide uniform controls that allow users to prioritize or proportionally allocate processor cycles across all applications, regardless of whether or not they have real-time requirements.

SMART achieves this behavior by reducing this complex resource management problem into two decisions. One based on importance determines the overall resource allocation for each activity. The other based on urgency determines when each activity is given its allocation. SMART provides a common importance attribute for both real-time and conventional activities based on priorities and weighted fair queueing. SMART then uses an earliest-deadline urgency mechanism to order when activities are serviced, allowing real-time activities to make the most efficient use of their resource allocations to meet their time constraints. A bias on batch activities accounts for their ability to tolerate more varied service latencies. This improves performance for interactive and real-time activities during periods of transient overload.

We have implemented SMART in a commercial operating system and measured its performance against other schedulers in executing applications with and without real-time requirements, including continuous media, interactive, and batch applications. Our results demonstrate SMART's ability to provide superior performance for multimedia applications.

To my parents

Hsi-Sheng and Loretta Nieh

# Acknowledgments

I would like to thank my advisor Monica Lam for her encouragement and unwavering support throughout this work. Her enthusiasm, boundless energy, and demand for excellence made much of this work possible. John Hennessy and Mendel Rosenblum served as members of my orals and reading committees and provided helpful comments on both the dissertation itself and life after dissertation as well. Simon Wong graciously agreed to serve as my orals chairman.

I would also like to thank Duane Northcutt, my supervisor at Sun Microsystems Laboratories. Duane provided an invaluable industry perspective and focused my efforts during the formative stages of this work. Thanks are due to Jim Hanko, Jerry Wall, and Alan Ruberg, for many enlightening discussions during my time at Sun Labs. I owe particular thanks to Jim for answering numerous questions about kernel internals, and for providing many of the applications used in this research.

I have been privileged to work with talented graduate students at Stanford. Brian Schmidt created tools that greatly simplified the processing of the experimental data, and conducted media synchronization experiments that served as a precursor to some of our audio/video synchronization work. He was a great officemate and a pleasure to work with. Amy Lim is a dear friend who read many drafts of the refereed papers drawn from the material in this dissertation.

I was fortunate to make good friends during my time at Stanford, who enriched my graduate years. Thanks to David Chen, Jolly and Liz Chen, Birk Lee, and Alice Yu for being a part of this journey. Special thanks to my best friend and wife Belinda, who God brought into

my life during these years. Her constant companionship and support when it was most needed were a tremendous blessing.

This dissertation is dedicated to my parents. I am grateful for their sacrifices that have made this education possible. And thanks be to Jesus, whose sacrifice gives life meaning, and whose grace has been abundant inspite of my shortcomings.

This work was supported in part by an NSF Young Investigator Award and Sun Microsystems Laboratories.

# Table of Contents

# List of Tables

# List of Figures

# 1 Introduction

The workload on computers is rapidly changing. In the past, computers were used in automating tasks around the work place, such as word and accounts processing in offices, and design automation in engineering environments. The human-computer interface has been primarily textual, with some limited amount of graphical input and display. With the phenomenal improvement in hardware technology in recent years, even highly affordable personal computers are capable of supporting much richer interfaces. Images, video, audio, and interactive graphics have become common place. A growing number of multimedia applications are available, ranging from video games and movie players, to sophisticated distributed simulation and virtual reality environments. In anticipation of a wider adoption of multimedia in applications in the future, there has been much research and development activity in computer architecture for multimedia applications. Not only is there a proliferation of processors that are built for accelerating the execution of multimedia applications, even general-purpose microprocessors have incorporated special instructions to speed their execution [29].

While hardware technology has advanced to support the special demands of multimedia applications, software environments have not. In particular, multimedia applications such as those that manipulate digital audio and video data often have application-specific timing requirements associated with their execution. For instance, to provide smooth playback of a video sequence at a standard 30 frames/sec rate, successive video frames need to be displayed within 33 ms of each other. However, today's general-purpose operating systems are not effective in supporting these real-time requirements. These multiprogrammed systems were designed with an emphasis on fairness and throughput without regard to meeting specific timing requirements. With the growing number of multimedia applications in the general-purpose computing environment that do in fact have real-time requirements that need to be met to function correctly, we are now faced with a growing resource management problem. Addressing this resource management problem is the topic of this dissertation.

A fundamental task of any operating system is the effective management of the system's resources. Resources that must be properly managed include processor cycles, virtual and physical memory, and I/O bandwidth. Although mismanagement of any of these resources can lead to poor system function [70], we have focused on processor scheduling in this dissertation. The question that we address is: given today's multitasking multimedia environments in which users may run several competing applications at once, how does the processor scheduler decide which application should run and when it should run to deliver the best performance? Processor cycles are often highly oversubscribed, with many applications being able to consume much more processing power than can be provided. In such an environment, the degree of effectiveness of processor scheduling is a dominant factor in overall system performance.

Our interest in this problem began when we noticed severe performance problems when users executed a mix of interactive, batch, and multimedia applications on a popular commercial desktop computer system. We observed pathological behaviors in which the video would freeze and the system would even stop accepting user input. Through careful measurement of actual application and operating system performance, we were able to isolate the performance problems and attribute them in large part due to poor processor scheduling by the operating system. Some improvements were made to the scheduler and incorporated into the next commercial release of the operating system to help alleviate the problems, but we realized that this stop-gap measure would not be enough. As a result, we started work on a novel approach to scheduling to address the problems that we were seeing in commodity operating systems. This gave birth to SMART, a Scheduler for Multimedia And Real-Time. The design, implementation, and evaluation of SMART are presented in this dissertation.

## 1.1. What is Multimedia?

Before going into further detail about the problems we encountered with existing systems in supporting multimedia applications and how SMART addresses them, let us take a step back to consider the nature of multimedia. For many people, multimedia is simply being able to watch a movie or play an audio CD on the computer at the touch of a button. However, multimedia is far more than just being able to display a stream of video or a stream of audio. Multimedia is about being able to do computing with media. Much more interesting multimedia applications already exist today:

- *ShowMeTV* [67]. Technical discussions often revolve around a whiteboard, which limits such interactions to users in the same geographic proximity. ShowMeTV from Sun Microsystems, Inc. employs audio, video, and interactive graphics to create a virtual whiteboard. Each user is seated in front of a desktop computer equipped with audio and video input devices and the ShowMeTV application. Users can then join a virtual discussion, at which point video and audio input from the user is available to other users participating in the discussion. In addition, a shared whiteboard appears on the computer screen of each participant, on which participants can scribble and share ideas in much the same way they would do around a physical whiteboard. Such an application facilitates collaborative discussions in ways that were not previously possible, providing participants separated by distance with a rich communication medium accessible from the convenience of their desktops.

- *Joke Browser* [2]. David Bacher developed the Joke Browser while at MIT as a demonstration of the benefits of using closed-captions for content-based media processing. It records late-night talk show monologues, including audio, video, and closed-captioned text. The Joke Browser analyzes the closed-captioned text to segment the monologues into jokes. A client application can be used to find jokes on a certain topic that have been made in the last week, with searching being done using the closed-captioned information. Jokes are recalled in their original multimedia format, with audio and video presentation, displaying all the quirks and mannerisms expected of a David Letterman show. Similar content-based media processing technology can be used to index sports highlights or search through network news programming.

- *NetCam* [49]. The job of a security guard is often filled with the tedium and monotony of monitoring a set of small monitors that display video input from video cameras located at the entrances to a property being monitored. Much of the time, such video footage consists of scenes of still life background. Only once in a while is there any human activity that would warrant a closer look. It would be much better if there were some way to eliminate the need to monitor still life background and only focus on the scenes where there is some human activity worth monitoring. Sun Microsystems, Inc. has created a product called NetCam which can be used exactly for this purpose. NetCam is a small network attached device that takes analog audio and video input, compresses it, and sends it over an existing local area network to be viewed on any computer console. In conjunction with a client application, NetCam can filter out audio and video footage of still life background so that only significant changes in the footage are displayed to the end user. The audio and video footage can be viewed from any networked computer console, subject to appropriate permissions. It can also be stored for later listening and viewing, which would allow a user at a later time to quickly browse through any interesting human activity. A video camera in the employee breakroom being processed by a NetCam might allow other employees to find out who always leaves the coffee pot empty, or who borrows the business section of the shared newspaper without returning it.

- *AV Photo Finder.* With the explosion of the World Wide Web, a slew of companies have created search engines that allow users to enter a text phrase and find web pages that contain information matching the respective text. This indexing and searching capability for text is being extended to images by AV Photo Finder, developed by Compaq's AltaVista and Virage, Inc. Given a text phrase, AV Photo Finder finds images from a database of images that pertain to the subject matter of the phrase. The database uses textual annotations assigned to each image to perform this content-based retrieval. In addition, image processing technology allows users to click on any of the images returned by AV Photo Finder to find other similar images in the database. That is, AV Photo Finder allows users to find images based on visual appearance or content.

These examples share a set of common attributes which are characteristic of the kinds of multimedia applications we see coming on the horizon. First, all of these examples employ several media together, not just audio or video. ShowMeTV alone makes use of audio, video, text, as well as graphics to facilitate distance collaboration. It is plain to see that the "multi" in multimedia connotes the use of many media, including audio and video as well as more traditional media such as text, graphics, and images. In fact, the term *multimedia*, as first used in 1962, simply meant the use or involvement of several media [48]. Second, these examples do more than just allow several media to co-exist independently. Instead, they use several media together in an integrated and coordinated fashion. For instance, the Joke Browser uses closed-captioned text to segment the audio and video from monologues, and then attempts to display the various media streams in a coordinated and synchronized manner. Third, these examples marry computing and media to enable the processing of multimedia, not just the display of multimedia. For instance, NetCam analyzes the video input for significant scene changes to determine whether or not the video should be displayed. The computer is used to perform analysis on the media and take actions based on that analysis. Instead of being just a display device, the computer is an active participant that can be programmed to manipulate and control media itself.

## 1.2. Requirements of Multimedia on Processor Scheduling

To realize the full potential of multimedia, we believe that the ability to use software to process media is of key importance. All of the previous application examples rely on the ability to program computers to manipulate and control media. This software-oriented perspective of multimedia necessitates that the processor play a central role. Afterall, if the processor is not manipulating the media bits, how is an application going to do any real media processing?

Some may note though that there are a number of commercial products which deliver some amount of audio or video functionality to a desktop computer without using the processor to manipulate the media streams. For instance, a number of vendors manufacture a dedicated hardware device to allow a user to watch video on his computer by taking analog video input, bypassing the processor, and displaying the video directly to the screen. While such devices provide some video hardware acceleration, they do not provide true multimedia functionality. Such devices typically take just a single media input, providing only "unimedia" functionality. Just trying to support a NetCam-like security application would require more devices than there are slots available in many desktop computers. Typically with such hardwired devices, the only integration of their media stream input with other media is limited to sharing the same computer screen. Perhaps more importantly, these devices allow little if any interesting processing of the media before it is displayed because they completely bypass the processor. Reducing your desktop computer to a video display device is a far cry from the innovative applications possible by taking full advantage of a general-purpose programmable computing environment.

If it were the case that users were satisfied with running one multimedia application at a time, the need for processor scheduling would be greatly reduced. However, users have come to expect that their multiprogrammed computers should be capable of running multiple applications at once, often being run by multiple users. We expect no less of this multitasking functionality in running multimedia applications. We want to be able to have NetCam running in the background while we are participating in a ShowMeTV discussion about some recent monologue from the Joke Browser. We want to be able to have network news programming running in the background while being able to compile a program and surf the World Wide Web. When multiple applications need to be run on the same processor, scheduling decisions need to made regarding how the processor is shared and when each application gets to run. With the processor playing a central role in the processing of multimedia, doing an effective job of processor scheduling is crucial for delivering good system performance.

To understand the requirements imposed by multimedia applications on processor scheduling, we first describe the salient features of multimedia applications and the computing environment in which they execute. We contrast these characteristics with those of traditional applications that current operating systems are designed for. In particular, we focus on the temporal characteristics of multimedia applications and their impact on scheduler design.

To aid our discussion, we define some terms here that are used in this dissertation. We use the term *activity* to denote a schedulable entity. An application may consists of one or more activities. An activity has a set of attributes associated with it that may change over time. In particular, an activity is called a *real-time* activity if the activity has some timing requirement associated with it. An activity that does not have such a timing requirement is non-real-time and is called a *conventional* activity. As it executes, an activity may change between being real-time or conventional in nature. We will loosely refer to an application as being real-time if it consists of one or more real-time activities. We will refer to an application as being conventional if it consists of no activities with real-time requirements.

### 1.2.1. Timing Characteristics

While there are many forms of media, they can be divided into two classes that have very different timing characteristics: *continuous* and *discrete*. Continuous media are time dependent. Discrete media are time independent. Examples of continuous media are digital audio and video. Examples of discrete media are text, still images, and graphics.

Unlike discrete media, continuous media have inherent timing requirements. A continuous media stream consists of a time sequence of media samples, such as audio samples or video frames. The distinguishing characteristic of such data is that information is expressed not only by the individual samples of the stream, but by the temporal alignment of the samples as well. For example, consider a captured video stream showing a ball bouncing up and down. The rate of motion of the bouncing ball is encoded in the time spacing between video frames. To accurately reproduce the motion of the bouncing ball when the video stream is displayed, the elapsed time between displayed frames should be the same as the elapsed time between the respective frames when they were captured.

Not only are there timing requirements within a continuous media stream, but there may also be timing requirements among multiple media streams as well. These timing requirements are often due to the need to synchronize multiple media streams. For instance, in playing a movie, the audio stream and the video stream need to be synchronized so that the desired audio is heard when a given video frame is displayed. Moreover, multimedia can require synchronization across continuous and discrete media, as in the case of closed-captioned television programming in which text, audio, and video are displayed together. Note that when a time independent media is synchronized with a time dependent media, the result is time dependent as well. This time dependent nature that arises from synchronizing media streams is a key characteristic of multimedia applications.

When an application processes and displays continuous media streams, it must typically meet two kinds of timing requirements to preserve the temporal alignment of the media streams being processed. One requirement is that any delay due to processing between the input media stream and the processed output media stream should be as constant as possible. Variance in the delay introduces undesirable jitter in the output stream. The other requirement is that the application must process media samples fast enough. If media samples are not processed at the rate at which they arrive, then they will be late being displayed and it will not be possible to maintain the exact temporal alignment of the media samples. Note that these timing requirements are typically soft real-time in nature as opposed to hard real-time. While the inability to process a media sample within its time constraints is often objectionable, being late only diminishes the quality of the results. It does not lead to a catastrophic failure, as would characterize a hard real-time requirement.

The timing requirements in processing continuous media are commonly cyclic in nature, though they may also be aperiodic. For many continuous media streams, the desired performance is to have the respective media samples displayed at evenly spaced intervals. For instance, the video frames in a video stream are typically displayed at 30 frames/sec, or one frame every 1/30th of a second. On the other hand, there are also times when the timing requirements are aperiodic in nature. For instance, the timing requirements in processing the video frames in a video stream may not be evenly spaced because there are frames missing from the video stream. Alternatively, aperiodic timing requirements can arise when an application is attempting to synchronize the display of a media stream with some form of user interaction, which is commonly aperiodic.

### 1.2.2. System Load Characteristics

When executing multimedia applications, the resulting load on the system is often high and very dynamic. Multimedia applications present practically an insatiable demand for resources. Even with the rapid advances in hardware technology, today’s workstation-class computers are just beginning to be able to use software techniques to display full resolution (640x480 pixels) video at full frame rate (30 frames per second). Emerging HDTV video standards will require much more computational power. Doing interesting processing with video will require even more computational power. The multimedia applications that we see on the horizon will only further consume already insufficient processor cycles. As applications such as real-time video are highly resource intensive and can consume the resources of an entire machine, resources are commonly overloaded, with resource demand exceeding its availability.

Much of the work that has been done to support real-time requirements has been in the context of embedded real-time systems in which the application timing requirements and the execution environment are static and strictly periodic in nature. In contrast, the general-purpose computing environment in which multimedia applications execute is highly dynamic in nature. Users may start or terminate applications at any time, changing the load on the system.

The processing requirements of multimedia applications themselves are often highly dynamic as well. While the media samples in continuous media streams typically occur in time in a periodic manner, the processing requirements for the media samples are often far from being periodic. For instance, the processing time to uncompress or compress JPEG or MPEG encoded video can vary substantially for different video frames. Alternatively, the processing requirements of a multimedia application may vary depending on how it is being used. For example, in the case of a movie player application, the processing time requirements of the application when it is fast forwarding through a movie will be quite different from when it is doing normal playback.

### 1.2.3. Adaptive Characteristics

As multimedia applications are highly resource intensive, even a single full-motion full-resolution video application can often consume the resources of an entire machine. Recognizing that the system may lack sufficient resources to meet the timing requirements of all multimedia applications, these applications are often able to adapt by offering different qualities of service depending on resource availability. They can tradeoff the quality of their results versus the consumption of processing time. In the case of video for instance, if a video frame cannot be displayed within its timeliness requirements, the application might simply discard the video frame and proceed to the next one. If many of the frames cannot be displayed on time, the application might choose to discard every other frame so that the remaining frames can be displayed on time. Alternatively, a video application may be able to reduce the picture quality of each frame to reduce its processing requirements so that each frame can be displayed on time.

### 1.2.4. Mixed Mode Characteristics

While there is a need to support real-time activities such as those found in multimedia applications, it is important for the system to continue to be able to run existing conventional applications effectively. In particular, real-time activities should not always be allowed to run in preference to all other activities because they may starve out important conventional activities, such as those required to keep the system running.

Not only are there real-time and conventional applications that must be able to run together on the same system, but multimedia applications may mix continuous and discrete media processing within a given application. For instance, part of the goal of multimedia is to be able to create interesting multimedia documents that mix time independent media such as text and graphics with time dependent media such as audio and video. This results in the execution of both real-time and conventional activities.

Real-time and conventional activities must be able to co-exist and share resources. In no way should the capabilities of a multiprogrammed general-purpose computer be reduced to a single function system, be it a commodity television set or other embedded system, in order to meet the demands of multimedia applications.

### 1.2.5. User Characteristics

Different users may have different preferences for how a mix of applications should behave. For instance, when a system is overloaded due to the Joke Browser and Show-MeTV running simultaneously, one user may want to reduce the quality of the Joke

Browser video to free up resources that can be used to improve the display quality of the ShowMeTV. Another user may desire the opposite behavior. The desired behavior may depend on whether the user is actively engaged in a ShowMeTV collaborative session, or if the user is bored and would prefer to be entertained for the moment. Alternatively, how a system partitions resources to trade off the speed of a compilation versus the display quality of a video depends on which of the two applications is more important to a user. It may depend on whether the video is part of an important teleconferencing session or just a television show being watched while waiting for an important computational application to complete. Not only may applications be of varying importance to a user, but different users may be considered to be of differing importance for a shared computer system. For instance, the owner of a given workstation would typically have priority for the computing resources of that workstation over other users. Alternatively, a computing service provider might give higher priority for computing resources to customers that paid a higher service fee. It is desirable for a system to be flexible enough and have a wide enough range of behavior to allow different users to obtain different application mix behavior based on their preferences.

### 1.2.6. Summary

Historically, real-time computing systems have focused on meeting the requirements of activities with time constraints, while general-purpose computing systems have focused on meeting the requirements of conventional activities in the context of a highly dynamic computing environment. The advent of multimedia has brought these activities together in a new way, along with diverse requirements that arise from mixing two very different kinds of activities. Unlike traditional real-time or general-purpose computing systems, systems that support multimedia must support the mix of real-time and conventional activities in a manner that allows real-time activities to meet their timing requirements without causing the starvation of important conventional activities. Moreover, such systems will be expected to provide this support in the context of a user-centric, highly dynamic, frequently overloaded, general-purpose computing environment.

## 1.3. Current Practice and Research

In today's general-purpose computing environment, there is little operating system support for the requirements of multimedia applications. Multimedia applications that are written today have little choice but to rely on traditional timesharing operating systems which dominate the general-purpose computing infrastructure. As a result, these applications must provide their own internal framework for dealing with timing requirements. To run effectively, they often rely on being able to monopolize the system, or require that the system being used have excess resources. In this way, poor scheduling is less likely to result in frequently missed deadlines. However, these workarounds fly in the face of user expectations to be able to use multimedia effectively in a fully functional multitasking environment. Furthermore, the fact that many multimedia applications are highly dynamic and highly resource intensive means that relying on excess resources can require much more expensive hardware than would otherwise be necessary with more effective resource management. With commercial computer vendors primarily concerned with the price-performance of their systems, there needs to be a better way.

Anticipating that processor scheduling based on traditional timesharing would not be suitable for the support of multimedia applications, commercial computer vendors have incorporated into their operating systems features designed to support applications with real-time requirements. In particular, UNIX System V Release 4 (SVR4) [68] provides a real-time static priority scheduler, in addition to a standard UNIX timesharing scheduler. By scheduling real-time activities at a higher priority than any other class of activities, UNIX SVR4 allows real-time activities to obtain processor resources when needed in order to meet their timeliness requirements. This solution claims to provide robust system support for multimedia applications by allowing applications such as those that manipulate audio and video to serviced by the real-time scheduler. Not only is UNIX SVR4 the most common basis of UNIX operating systems, but a similar priority-based scheduling framework is used in many other operating systems, including Windows NT [11]. Through careful measurements of application performance, we have quantitatively demonstrated that the UNIX SVR4 scheduler manages system resources poorly, resulting in unacceptable system performance for multimedia applications. Not only are the application latencies much worse than desired, but pathologies occur with the scheduler such that the system no longer accepts user input [50].

Because of the importance of effective processor scheduling for multimedia, a number of different scheduling approaches have been proposed to attempt to address this resource management problem. One common approach is resource reservations. The basic idea with reservations is that each real-time activity is allowed to reserve a fixed percentage of the processor for meeting its timing requirements. For each real-time activity that has been assigned a reservation for processor cycles, the scheduler then uses some form of real-time scheduling to meet the timing requirements of the activities. Conventional activities are typically allowed to share any unreserved processor cycles in a time-sharing fashion. A primary limitation of this approach is that assigning the reservations to activities in any meaningful way remains an open problem that is currently at best a trial-and-error procedure. In addition, because conventional activities are given leftover processor cycles, their starvation is a real possibility.

Another common approach is fair queuing. The basic idea with fair queuing is that each activity is assigned some number of shares. The scheduler then allocates processor cycles among the activities in proportion to their shares. As the system load and the number of running activities change over time, there is no need to adjust the assignment of shares to ensure that each activity is given its proportional allocation of processor cycles, since it is the ratio of the shares that matters. While fair queuing provides a flexible resource allocation abstraction, it does not at the same time schedule real-time activities effectively in meeting their timing requirements.

Because of the difficulty of scheduling across both real-time and conventional activities, many researchers have turned to hierarchical scheduling. This is a divide-and-conquer approach in which each class of activity can be assigned its own scheduling policy. There can be a real-time scheduling policy tailored to the needs of real-time activities and a conventional scheduling policy tailored to the needs of conventional activities. However, there still needs to be some way of merging these policy decisions together and the manner in which this is done is a major factor in the overall effectiveness of the scheduler. Typically, the policies are merged together using priorities or proportional sharing. The problem is that using priorities results in the problems encountered with UNIX SVR4, while using proportional sharing results in the problems encountered with fair queuing.

## 1.4. The SMART Approach

To address these problems, this dissertation proposes SMART (Scheduler for Multimedia And Real-Time applications), a processor scheduler  that fully supports the requirements of multimedia applications. SMART consists of a simple application interface and a scheduling algorithm that tries to deliver the best overall value to the user. For applications with real-time requirements, SMART explicitly accounts for application-specific timing information in making scheduling decisions and employs a deadline-based scheduling algorithm. SMART is able to make efficient use of resources in meeting real-time requirements even under dynamically changing system loads, including when the system is overloaded. It can provide feedback to applications regarding the availability of resources to allow applications to adapt based on the system load. In addition, the support for real-time applications is integrated with the support for conventional applications. This allows the user to prioritize the sharing of resources across real-time and conventional applications, or proportionally share resources across both kinds of applications, irrespective of their timing requirements. As the system load changes, SMART adjusts the allocation of resources dynamically and seamlessly in accordance with user preferences.

SMART achieves this behavior by reducing this complex resource management problem to two decisions, one based on *importance* to determine the overall resource allocation for each activity, and the other based on *urgency* to determine when each activity is given its allocation. SMART provides a common importance attribute for both real-time and conventional activities based on priorities and weighted fair queueing (WFQ) [12]. SMART then uses an urgency mechanism based on earliest-deadline scheduling [45] to optimize the order in which activities are serviced to allow real-time activities to make the most efficient use of their resource allocations to meet their time constraints. In addition, a bias on conventional batch activities that accounts for their ability to tolerate more varied service latencies is used to give interactive and real-time activities better performance during periods of transient overload.

We have implemented SMART in the Solaris operating system, a UNIX SVR4 commercial operating system developed by Sun Microsystems. To evaluate this system, we have measured its performance in a full-function system environment on real applications. We have also quantitatively compared SMART's performance against other schedulers commonly used in both research and practice. We believe that measuring real applications in a real system is the best way to understand and evaluate scheduling performance, especially in view of the complex requirements of multimedia applications. While most multimedia application studies focus exclusively on audio and video applications, our studies include experimental results for real-time audio and video, interactive, and batch applications. We believe it is important to understand the interactions of different classes of applications and provide good performance for all classes of applications. Our results show that SMART (1) delivers optimal performance for real-time activities when the system is underloaded, (2) can deliver almost a factor of two better performance than schedulers used in practice and research in meeting real-time requirements when the system is overloaded, (3) delivers good interactive responsiveness and batch computational performance while effectively meeting real-time requirements, (4) provides flexible proportional and prioritized resource sharing across both real-time and conventional activities, (5) and provides predictable controls that are well correlated with experimentally measured application behavior.

## 1.5. Dissertation Overview

This dissertation is organized as follows. Chapter 2 describes the quantitative measurements that demonstrate the limitations of commodity operating systems in supporting multimedia applications. Chapter 3 begins our discussion of SMART, starting with the SMART interface. Chapter 4 presents the SMART scheduling algorithm. Chapter 5 describes issues in the implementation of SMART in a commercial operating system. Chapter 6 presents experimental results that quantify the performance benefits of SMART as compared with other approaches. Finally, we present some conclusions and directions for future work.

# 2 Limitations of Commodity Operating Systems

Commodity multiprogrammed operating systems, typified by the UNIX operating system, evolved from the much different environment of large-scale, multi-user, time-sharing systems. These time-sharing systems attempt to be fair to all applications while maximizing total system throughput. Without explicit information from the applications, the best that can be done is to use heuristics that intuit some properties of applications from their behavior and adjust the allocation of resources in a way that provides the desired results. For example, some schedulers attempt to approximate a shortest-processing-time-first algorithm by observing the accumulated run-time of processes and giving fewer processing cycles to those tasks which run for longer periods of time [39]. Such heuristics may result in less than desirable behavior when users want to run long-running multimedia applications that are both compute-intensive and have tight timing requirements that need to be met.

In an attempt to support applications with real-time requirements in the context of a general-purpose computing environment, AT&T's UNIX System V Release 4 (SVR4) was designed to include a real-time static priority scheduler, in addition to a standard UNIX time-sharing scheduler [68]. A user can indicate to the operating system that an application has timing requirements that need to be met by classifying the application as a real-time job and assigning it a priority value. UNIX SVR4 ensures that all real-time activities are assigned strictly higher priorities than that of any other class of activities. By running all activities in priority order, UNIX SVR4 allows real-time activities to obtain processor cycles when needed to meet their timing requirements. This solution claims to provide robust system support for multimedia applications by allowing applications such as those that manipulate audio and video to be serviced by the real-time scheduler. Since UNIX SVR4 is a common basis of commercial operating systems, it is important to investigate these assertions. Therefore, we have used a UNIX SVR4 based system to examine actual performance of real multimedia applications running in a workstation environment.

Through careful measurements of application and system software performance, we quantitatively demonstrate that the UNIX SVR4 scheduler manages system resources poorly for both so-called real-time and conventional activities, resulting in unacceptable system performance for multimedia applications. Not only are the application latencies much worse than desired, but pathologies occur with the scheduler such that the system no longer accepts user input. To alleviate some of these problems, a number of modifications were made to the UNIX SVR4 time-sharing scheduler. While the modified time-sharing scheduler is not very effective, these modifications do result in a noticeable performance improvement for multimedia applications. The time-sharing scheduler in Sun's Solaris operating system, version 2.3 and later, is based on the modifications described in this work.

This chapter describes experiments and measurements we performed to identify the limitations of commodity operating systems in supporting multimedia applications. It is organized as follows. Section 2.1 provides an overview of the experiments. Section 2.2 describes the experimental setup and applications that we used for our measurements. Section 2.3 presents our measurements. Section 2.4 discusses the results. We then present some summary remarks regarding the limitations of commodity operating systems in supporting multimedia applications.

## 2.1. Overview of Experiments

To examine the ability of the processor scheduling policies of UNIX SVR4 to support multimedia applications, we identified three classes of computational activities that characterize the main types of programs executed on workstations: interactive, continuous media, and batch. Interactive activities are characteristic of applications (e.g., text editors or programs with graphical user interfaces) in which computations must be completed within a short, uniform amount of time in order not to disrupt the exchange of input and output between the user and application. Continuous media activities are characteristic of applications that manipulate sampled digital media (e.g., television or teleconferencing) within application-specific timing requirements. Simple continuous media activities are often cyclic computations that process and transport media samples at a defined rate. Batch activities are characteristic of applications (e.g., long compilations or scientific programs) in which the required processing time is sufficiently long to allow users to divert their attention to other tasks while waiting for the computation to complete. By selecting applications from each of these classes, a representative workload can be constructed that characterizes typical multimedia workstation usage. To simplify the experiments and the task of interpreting the resulting data, only one program from each class is used in the following experiments.

To obtain valid results, the experimentation was done with a standard, production workstation and operating system. However, measurements of actual system behavior are quite complex as compared to simulation-based experimentation. As a result, as we discuss in Section 2.2, a number of measures were taken to permit repeatability of experimental results and allow the identification and isolation of processor scheduling effects. Since the purpose of the experiments is to explore the effectiveness of various processor scheduling policies, an attempt was made to minimize the effects of other resource management decisions. Results were collected from the execution of a series of trial runs of the representative programs on the testbed hardware. The parameters of the trials were chosen so as to permit the exploration of a wide range of different conditions with the minimum number of experiments.

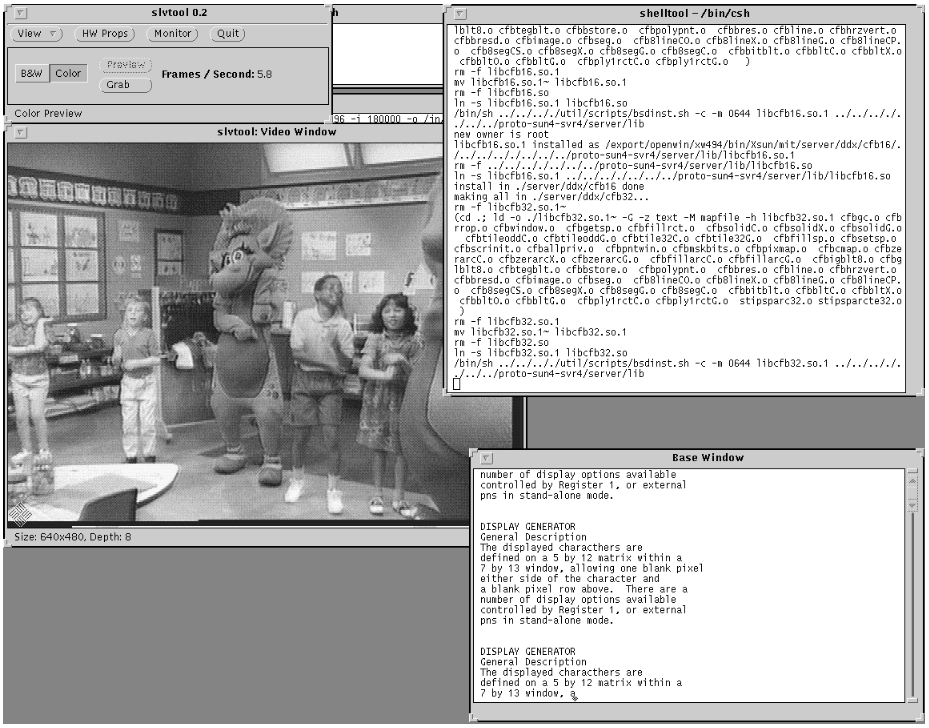
## 2.2. Experimental Design

To characterize typical workstation usage, three applications were chosen to represent interactive, continuous media, and batch activities. A sample screen shot of these applications is shown in Figure 2-1. Each of these programs was implemented in the most obvious, and straight-forward fashion. The applications were:

- *typing* (interactive class) — This application emulates a user typing to a text editor by receiving a series of characters from a serial input line and using the X window server [60] to display them to the frame buffer.

- *video* (continuous media class) — This is a real-time video player application (e.g. as used for television, teleconferencing) that attempts to show frames of video at a constant rate. *Video* captures data from a digitizer board, dithers to 8-bit pseudo-color, and relies on the X window server to render the pixels to the frame buffer. Video frames are 640x480 pixels.

- *compute* (batch class) — This application is intended to represent programs such as the UNIX *make* utility. *make* execution is characterized by repeated spawning and waiting for various programs such as compiler passes, assemblers, and linkers. To reduce variability induced by the system's virtual memory, file system, and disk I/O handling, a simple shell script was used that repeatedly forks and waits for small processes to complete (in this case, the UNIX *expr* command).



Frames / Second: 5.8

Size: 640x480, Depth: 8

```
lblt8.o cfbtegblt.o cfbbstore.o  cfbpolypnt.o  cfbbres.o cfbline.o cfbhrzvert.o
cfbbresd.o cfbimage.o cfbseg.o  cfb8lineCO.o cfb8lineX.o cfb8lineG.o cfb8lineCP.
o  cfb8segCS.o cfb8segX.o cfb8segG.o cfb8segC.o  cfbbitblt.o cfbbltC.o cfbbltX.o
 cfbbltO.o cfbbltG.o  cfbply1rctC.o cfbply1rctG.o   )
rm -f libcfb16.so.1
mv libcfb16.so.1~ libcfb16.so.1
rm -f libcfb16.so
ln -s libcfb16.so.1 libcfb16.so
/bin/sh ../../.././util/scripts/bsdinst.sh -c -m 0644 libcfb16.so.1 ../../.././.
./../../proto-sun4-svr4/server/lib
new owner is root
libcfb16.so.1 installed as /export/openwin/xw494/bin/Xsun/mit/server/ddx/cfb16/.
/../../.././../../../proto-sun4-svr4/server/lib/libcfb16.so.1
rm -f ../../.././../../../proto-sun4-svr4/server/lib/libcfb16.so
ln -s libcfb16.so.1 ../../.././../../../proto-sun4-svr4/server/lib/libcfb16.so
install in ./server/ddx/cfb16 done
making all in ./server/ddx/cfb32...
rm -f libcfb32.so.1~
(cd .; ld -o ./libcfb32.so.1~ -G -z text -M mapfile -h libcfb32.so.1 cfbgc.o cfb
rrop.o cfbwindow.o  cfbgetsp.o cfbfillrct.o  cfbsolidC.o cfbsolidX.o cfbsolidG.o
  cfbtileoddC.o cfbtileoddG.o cfbtile32C.o cfbtile32G.o  cfbfillsp.o cfbsetsp.o
cfbscrinit.o cfballpriv.o  cfbpntwin.o cfbmskbits.o cfbpixmap.o  cfbcmap.o cfbze
rarcC.o cfbzerarcX.o cfbzerarcG.o  cfbfillarcC.o cfbfillarcG.o  cfbigblt8.o cfbg
lblt8.o cfbtegblt.o cfbbstore.o  cfbpolypnt.o  cfbbres.o cfbline.o cfbhrzvert.o
cfbbresd.o cfbimage.o cfbseg.o  cfb8lineCO.o cfb8lineX.o cfb8lineG.o cfb8lineCP.
o  cfb8segCS.o cfb8segX.o cfb8segG.o cfb8segC.o  cfbbitblt.o cfbbltC.o cfbbltX.o
 cfbbltO.o cfbbltG.o  cfbply1rctC.o cfbply1rctG.o  stipsparc32.o stipsparcte32.o
 )
rm -f libcfb32.so.1
mv libcfb32.so.1~ libcfb32.so.1
rm -f libcfb32.so
ln -s libcfb32.so.1 libcfb32.so
/bin/sh ../../.././util/scripts/bsdinst.sh -c -m 0644 libcfb32.so.1 ../../.././.
./../../proto-sun4-svr4/server/lib
```

```
number of display options available
controlled by Register 1, or external
pns in stand-alone mode.


DISPLAY GENERATOR
General Description
The displayed characthers are
defined on a 5 by 12 matrix within a
7 by 13 window, allowing one blank pixel
either side of the character and
a blank pixel row above.  There are a
number of display options available
controlled by Register 1, or external
pns in stand-alone mode.


DISPLAY GENERATOR
General Description
The displayed characthers are
defined on a 5 by 12 matrix within a
7 by 13 window, a
```

Figure 2-1. Sample application screen

A number of software tools were added to the testbed to permit the logging of significant events into files, and the post-processing of these files for the generation of tracing reports. Modifications were made to the application programs and components of the system software to generate the necessary tracing events, but these modifications did not measurably change the performance of the software.

While not strictly an application program, the X window server represents a fourth major component that contributes to the overall performance of the system in these experiments. It was necessary to instrument the window server to obtain the desired measurements of user-level system performance. In particular, the *typing* application displays the typed characters to the computer display via the X window server. To obtain measurements of the time between when a character was typed and when it was displayed, it is important to include measurements of the time taken by the X window server to display the character on behalf of the application. Similarly, the *video* application displays video to the computer display via the X window server. To obtain measurements of the time between when a video frame arrives and when it is displayed, it is important to include measurements of the time taken by the X window server to display the video frame on behalf of the application. Note that the window system's behavior *per se* is not of interest here, only its contribution to the user-visible performance of the application programs in the example mix.

The experiments were performed in a representative workstation environment; it consisted of a SparcStation10 with a single 50MHz processor and 64MB of primary memory. The testbed system included a standard 8bit (pseudo-color) frame buffer controller (i.e., GX), and a 1GB local (SCSI) disk drive. In addition, the testbed workstation began with the release of Sun's operating system that was in distribution prior to this work — Solaris 2.2 [18], which is based on UNIX SVR4.

UNIX SVR4 supports multiple concurrent scheduling policies, called *scheduling classes*. In particular, a real-time class (RT) class and a time-sharing (TS) class are included in UNIX SVR4. The scheduling classes are unified into a single priority scheduler by mapping each of them onto a range of global priorities, with time-sharing processes mapped to the low priority range and real-time processes to the highest priority range. UNIX SVR4 also provides a set of commands for assigning processes to a class and controlling each class. These were used to assign processes for each experiment to the RT class, the TS class, or to a new scheduling class we developed as described later. In addition, for some experiments, controls specific to the scheduling class were used to modify their default behaviors.

To support the *video* continuous media application, an SBus I/O adaptor was constructed and added to the system. The adaptor permits the decoding and digitization of analog video streams into a sequence of video frames. This video digitizing unit appears as a memory-mapped device in an application’s address space and allows a user-level application to acquire video frames, whose pixels can be color-space converted into RGB values, dithered to 8-bit depth, and displayed via the window system.

An effort was made to eliminate variations in the test environment to make the experiments repeatable. To this end, the testbed was disconnected from the network and restarted prior to each experimental run. In addition, to enable a realistic and repeatable sequence of typed keystrokes for programs of the interactive class, a keyboard/mouse simulator was constructed and attached to the testbed workstation. This device is capable of recording a sequence of keyboard and mouse inputs, and then replaying the sequence with the same timing characteristics.

## 2.3. Measurements

To evaluate a system’s performance, a means of measuring the system’s operation is needed that encompasses all of the activities in all of the applications. However, the measure of quality of an application’s performance is different for each class of application. To deliver the desired performance on interactive activities, the system should minimize the average and variance of time between user input and system response to a level that is faster than that which a human can readily detect. This means that for simple tasks such as typing, cursor motion, or mouse selection, system response time should be less than 50-150 milliseconds [62]. To deliver peak performance on simple display-oriented continuous media activities, the system should minimize the difference between the average display rate and the desired display rate, while also minimizing the variance of the display rate. In particular, uncertainty is worse than latency; users would rather have a 10 frames per second (fps) constant frame rate as opposed to a frame rate that varied noticeably from 2 fps to 30 fps with a mean of 15 fps [71]. To deliver good performance on batch activities, the system should strive to minimize the difference between the actual time of completion and the minimum time required for completion as defined by the case when the whole machine is dedicated to the given activity. In other words, if a *make* takes 10 minutes to complete on an unloaded system, the user would like the *make* to take $10\times(1+\delta)$ minutes, where $\delta$ is as small as possible, to complete even when there are other activities running on the system.

Because the relative value of each application to a user is subjective and application performance is measured in many different ways (i.e. interactive character latency verses video frame rate), no single figure-of-merit can be derived to compare test results. That is, any calculation resulting in a single value would require an assignment of weights and conversion factors to each measurement to account for the relative values of the applications and the different units of measurement. Since any such arbitrary assignment is suspect and is likely to obscure significant information, the outcome of each test is presented as a *value contour*. In a value contour, the achieved performance on each measurement is charted relative to a normative *baseline* value. If a single figure-of-merit is desired, it can be derived by assigning weights appropriate to the relative value of each application to the contour data.

Using value contours based on the mean and standard deviation of characteristic execution times, we capture the essential quality metric for each application class. The measured characteristic and baseline values are shown in Table 2-1 for each of the applications. To obtain these baseline values, each application was run in isolation on an otherwise quiescent workstation. Note, therefore, that when multiple applications are run simultaneously, it is not generally possible for all of them to reach 100% of the baseline value. The data from the experiments described in this paper, obtained from running these applications simultaneously, is shown in Table 2-2.

| Application | Measurement | Mean | Std. Dev. |
|---|---|---|---|
| Typing | Latency between character arrival and rendering to frame buffer | 38.5 msec | 15.7 msec |
| Video | Time between display of successive frames | 112.0 msec | 9.75 msec |
| Compute | Time to execute one loop iteration | 149.0 msec | 6.79 msec |

Table 2-1. Application baseline values

| Application / Scheduling Class | | | | Typing | | Video | | Compute | |
|---|---|---|---|---|---|---|---|---|---|
| **X** | **T** | **V** | **C** | $\chi$ **(msec)** | $\sigma$ **(msec)** | $\chi$ **(msec)** | $\sigma$ **(msec)** | $\chi$ **(msec)** | $\sigma$ **(msec)** |
| TS | TS | TS | TS | 42900.0 | 23800.0 | 2780.0 | 9300.0 | 150.0 | 16.0 |
| TS+20 | TS | TS | TS-20 | 49.6 | 26.4 | 117.0 | 17.9 | 3910.0 | 699.0 |
| TS+20 | TS | TS-5 | TS-20 | 41.8 | 17.9 | 529.0 | 1430.0 | 189.0 | 279.0 |
| TS+20 | TS | TS-10 | TS-20 | 44.0 | 18.5 | 174.0 | 619.0 | 412.0 | 896.0 |
| TS | TS | RT | TS | — | — | 1100.0 | 4810.0 | 243.0 | 415.0 |
| RT | TS | TS | TS | 26400.0 | 14400.0 | 4230.0 | 9350.0 | 150.0 | 22.9 |
| RT- | TS | RT+ | TS | — | — | 142.0 | 260.0 | — | — |
| RT+ | TS | RT- | TS | 42000.0 | 32900.0 | 112.0 | 8.1 | 8040.0 | 2870.0 |
| TS-M | TS-M | TS-M | TS-M | 46.0 | 19.1 | 177.0 | 48.3 | 496.0 | 114.0 |

| Legend | | | |
|---|---|---|---|
| X | The X Window System server | TS | SVR4 TS (time-sharing class) |
| T | The *typing* application | TS±*n* | SVR4 TS with nice of ±*n* |
| V | The *video* application | RT | SVR4 RT (real-time class) |
| C | The *compute* application | RT+ | SVR4 RT with higher priority |
| $\chi$ | Mean | RT– | SVR4 RT with lower priority |
| $\sigma$ | Standard Deviation | TS-M | Modified TS scheduling class |
| — : Application did not complete measured operation | | | |

Table 2-2. Individual experiment results for UNIX SVR4 scheduling classes

The data in Table 2-2 shows that the choice of scheduler has a tremendous impact on application performance. The quality metrics for all of the applications differ by more than two orders of magnitude across different schedulers. There are even a number of instances in which an application did not even complete the measured operation.

Figure 2-2 presents a set of value contours derived from this data. In each contour, the first two bars, labeled ‘$T_\chi$’ and ‘$T_\sigma$’, represent the mean and standard deviation, respectively, for *typing* character latency. These values are normalized to the baseline values such that a full size bar represents a mean or standard deviation of latency as small as on an otherwise idle system (i.e. a taller bar represents better performance). Similarly, the bars labeled '$V_\chi$' and '$V_\sigma$', represent the normalized mean and standard deviation of the time between display of successive frames for *video*. Finally, the bars labeled '$C_\chi$' and '$C_\sigma$', represent the normalized mean and standard deviation of the time taken by one iteration of *compute*. The following section provides a description of the scenarios represented by each and an analysis of these results.

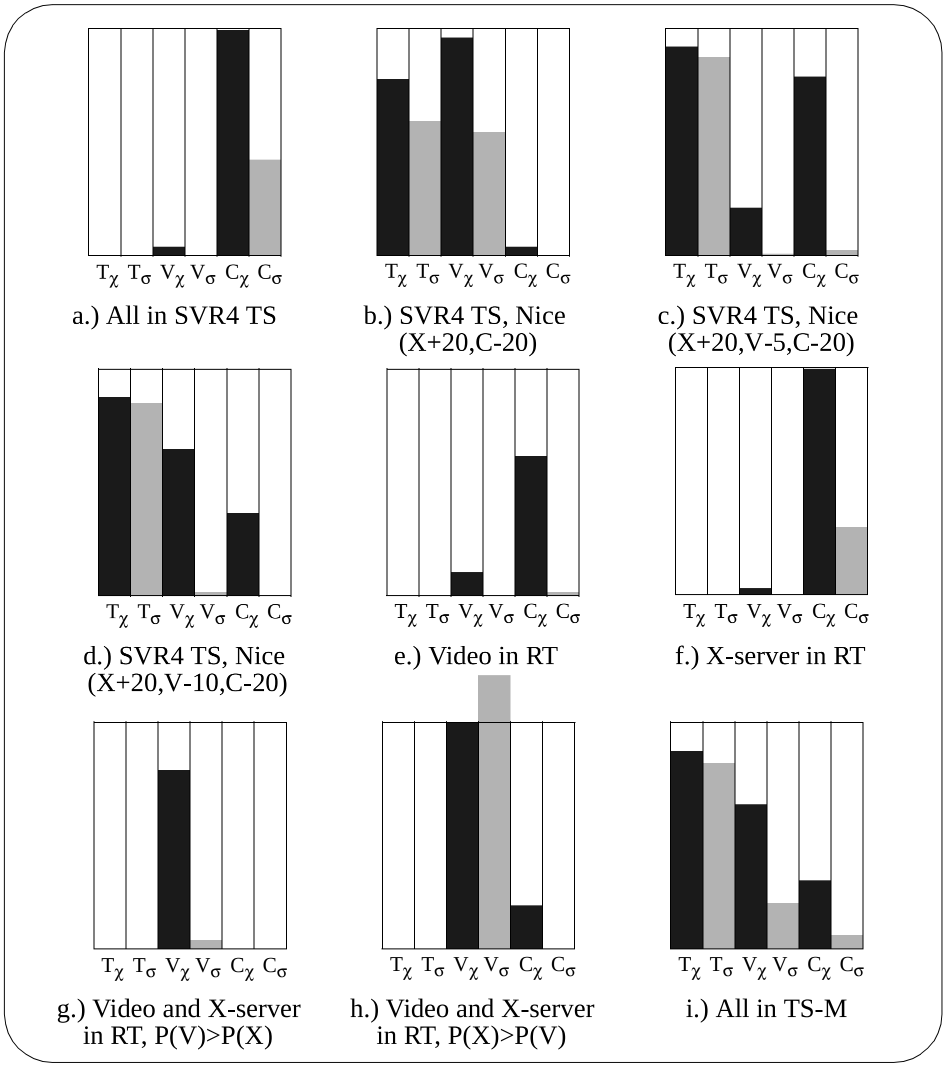


Figure 2-2. Application value contours

## 2.4. Interpretation of Results

It is expected that, in a well-behaved system, concurrent applications should all make some progress in their computation. That is, the running of an application by a user indicates some residual value for it. Therefore, no one application should be able to prevent others from running in absence of overt action by a user indicating this is the desired behavior. In addition, there should be no cases in which the system fails to respond to operator input; otherwise, control over the system is lost. Finally, users should be able to exercise a wide range of influence over the system's behaviors using a stable and predictable control mechanism.

The results of these experiments indicate that the standard UNIX SVR4 scheduling system often violates these objectives. The straightforward approach to adding multimedia applications to an SVR4-based workstation results, at best, in a low degree of value being provided to the users, and serious pathological behavior in the worst case. The following sections describe the test results for the SVR4 time-sharing class alone, the SVR4 time-sharing and real-time classes together, and a new implementation of the time-sharing class.

### 2.4.1. SVR4 Time-sharing Class

The first thing a typical user would do is simply run the chosen set of applications, which, by default, associates all applications with the time-sharing (TS) scheduling class. Doing this results in a pathological condition where the window system no longer accepts input events from the mouse or keyboard, causing the interactive application to freeze and the continuous media application to stop displaying frames of video. In fact, this pathology is so complete that attempts to stop the processes by typing commands in a shell (i.e. command interpreter) window prove futile, because the shell itself is not permitted to run.

The value contour for this scenario is shown in Figure 2-2a, and illustrates that all of the applications, with the exception of the batch job, contribute a relatively small amount to the total delivered value. This is due to the fact that the batch application forks many small programs to perform work, and then waits for them to finish. Because the batch application sleeps to wait for each child process to complete, the TS scheduling class identifies it as an I/O-intensive "interactive" job and provides it with repeated priority boosts for sleeping. As a result, the batch application quickly moves to the highest time-sharing priority value and remains there for the remainder of the experimental run.

An added effect occurs when the window server develops a backlog of outstanding service requests. As it works down this queue of outstanding commands, the TS scheduling class identifies the window server as CPU-intensive and lowers its priority. At the same time, because it sleeps in the process of obtaining new video frames, *video* is assigned a higher priority, allowing it to run and thereby generate additional traffic for the window server. As a result, the quality of the video being displayed is poor because the window system is not able to execute to process the frames fast enough. Worse yet, *typing* exhibits an average delay of more than 42 seconds from receiving a character to having it displayed, as opposed to the baseline value of 39 milliseconds. The interactive application suffers a degradation of three orders of magnitude because the window server, which must execute to render the character's pixels to the frame buffer, is not scheduled to run frequently enough to work its way through its growing backlog of commands. Moreover, due to the design of the standard SVR4 TS class, it can often take tens of seconds for the priority of a penalized process to recover to the point at which it can actually run. This augments the effect of the improper processor scheduling decisions and contributes to the poor overall performance of the system.

In an attempt to deal with this problem, the system's administrative controls were used to change the TS priorities of the window system and the applications. These user priorities are used by the TS scheduler to modify the actual scheduling priorities. These controls correspond roughly to traditional UNIX *nice* values. In one case, the user priority of the window system was elevated to the maximum possible level (+20), while the user priority of *compute* was depressed to the minimum possible level (–20), as shown in Figure 2-2b. This had the effect of improving the performance of *video* and *typing*, but *compute* barely ran. In an attempt to fix this, the user priority of *video* was degraded modestly (–5), resulting in the contour in Figure 2-2c. This shows how very small changes in these controls can lead to large and unpredictable effects. Finally, Figure 2-2d illustrates the result of *video* receiving a medium amount of degradation (–10). The achieved mean values of all applications are relatively high, but the variance in frame rate for *video* is unacceptably high. Note also the counterintuitive result that *video* performs better in this scenario than in Figure 2-2c, even though the scheduler controls indicated a lower importance for *video*.

Although the use of user priority adjustments could alleviate the pathological condition inherent in the SVR4 TS scheduling class, this approach is not effective in general (e.g., with multiple, independent applications). That is, it can take a great deal of experimentation to find a set of control values that work well, and the settings might only work for that exact application mix. In addition, this approach severely degrades the performance of *video*, resulting in highly variable display rates.

### 2.4.2. SVR4 Time-sharing and Real-time Classes

Although UNIX SVR4 also provides so-called “real-time” facilities, the assignment of different tasks to the real-time (RT) scheduling class yielded equally unsatisfactory results. Since *video* best fits the notion of what a real-time application is, the obvious first step for using the RT class is to assign *video* to it. However, when this is done, the system again ceases to accept input events from the mouse or keyboard and the video again degrades severely. This is due to the fact that any ready task in the RT class takes precedence over any TS task. Since *video* is almost always active, tasks in the TS class are hardly ever allowed to execute — in fact, shell programs are not even permitted to run, so a user cannot even attempt to stop such a “real-time” application. Once again, the quality of the video being displayed is poor because the window system is not able to execute to process the frames sent to it by the continuous media application. Again, the system delivers low overall value for any choice of value assignments, as shown in Figure 2-2e.

Alternatively, the window system could be associated with the RT class, with all of the applications remaining in the TS class. Although in such a case, the window system related activities (e.g., mouse tracking) perform well, the basic TS scheduling system pathology allows the batch job to monopolize the processor. As a result, none of the other applications can achieve even a small fraction of their possible value, as illustrated in Figure 2-2f.

Another attempt to provide a high degree of value to the user involves placing both *video* and the window system in the RT class, and having all applications remain in the TS class. In this case, the system executes *video* to the complete exclusion of all other processing. That is, neither *typing* nor *compute* are permitted to run at all, and it is not possible to type commands into the system’s shell windows. In fact, basic kernel services such as the process swapping, flushing dirty pages to disk, and releasing freed kernel memory are inhibited. The reason for this behavior is that *video* and the window server consume essentially all of the system's processor cycles, and real-time processes take precedence over all "system" and time-sharing processes. This is because the RT scheduler uses a strict priority policy, and no processes from other scheduling classes are permitted to run while there are ready processes in the RT class.

Figure 2-2f and Figure 2-2g show the results that are derived from placing the window system at a lower and at a higher RT priority than *video*, respectively. While neither case delivers acceptable results, the first case (i.e., with the window server's priority below *video*) was particularly bad because *video* did not leave sufficient time for the window server to process its requests. Note also, that in Figure 2-2h, *video* had less variance than in the baseline measurements. This is due to the strict priority scheduling discipline; processes in the RT class run in preference to all other processes, including system daemons.

Finally, we note that placing interactive applications in the RT class to improve their performance would also be ineffective unless the window server were placed in the RT class. Even then, proper operation is not assured because basic system services can be prevented from functioning due to resource demands in the higher priority real-time class. For example, when the X window server, *typing*, and *video* are run in the RT class, with priorities P(X)>P(*typing*)>P(*video*), *typing* unexpectedly performs more than three times worse than its baseline because it relies on streams I/O services [68] for character input processing. Because the streams processing is not done in the RT class, it is deferred in favor of the applications in RT, which consume virtually all of the CPU cycles.

### 2.4.3. Modified Time-sharing Class

A modified time-sharing (TS-M) scheduling class was developed to correct the problems demonstrated in these experimental runs. In particular, the modified version removes the anomalies of identifying batch jobs as interactive, and vice versa. In addition, it attempts to ensure that each process that can run is given the opportunity to make steady progress in its computation, while retaining a bias in favor of interactive processes. Finally, it reduces the feedback interval over which CPU behavior is monitored and penalties and rewards given.

The time-sharing scheduling class contained in Sun’s Solaris operating system, version 2.3 and later, is based on this work.

The results of the default use of this class for all applications and the window server process are given by Figure 2-2i. As can be seen, this delivers significantly better results for the continuous media and interactive applications than any combination of the standard SVR4 scheduling classes. It should also be noted that this scheduling policy achieves this level of performance without significantly starving the batch application, which still receives approximately 30% of the available CPU time.

Additional tests were performed by adjusting user priorities and by combining this new scheduling class with the SVR4 RT class (as was done with SVR4 TS class). However, with the exception of the cases where there was sufficient load in the RT class to consume all CPU cycles and starve the TS-M scheduling class, this resulted in no pathologies and showed a more predictable relationship between user priorities and application performance than the standard SVR4 TS scheduling class.

## 2.5. Summary

Through trial and error, it may be possible to find a particular combination of priorities and scheduling class assignments to make the SVR4 scheduling pathologies go away. However, such a solution would be extremely fragile and would require discovering new settings for any change in the mix of applications. In fact, these problems have been induced in many instances with different applications and conditions than those described here. For example, the continuous media application by itself can freeze the system when a user simply uses a popup menu. The modified time-sharing scheduling class eliminates these pathologies and provides default resource management behavior that favors interactive applications while not overly penalizing others.

Commodity operating systems, typified by UNIX SVR4, evolved from the much different environment of large-scale, multi-user, time-sharing systems. These systems attempt to be fair to all applications while maximizing total system throughput. As a result, a user (or system administrator) has only limited control over UNIX operating system resource management decisions.

Without such control it is not possible to provide the full range of behaviors that might be desired of multimedia applications. For example, providing uniform rates of audio and video presentation, where variance in the delivery rate is minimized, may be more important to some applications than others. Knowledge of the “slack” available in such computations can lead to more effective resource utilization. In addition, when the system is overloaded with continuous media applications, a way of identifying applications of lesser or greater importance to the users can allow the system to automatically perform service trade-offs rather than forcing it to degrade all applications equally at best, or randomly at worst. Armed with such information, the system can manage its resources in such a way as to maximize the total value delivered to the end user.

Finally, note that the existence of the strict-priority real-time scheduling class in standard UNIX SVR4 in no way allows a user to effectively deal with these types of problems. In addition, it opens the very real possibility of runaway applications that consume all CPU resources and effectively prevent a user or system administrator from regaining control without rebooting the system.

# 3 The SMART Interface and Usage Model

A fundamental task of an operating system is to manage the resources of a computer system in a predictable and reliable way that satisfies the needs of users. However, different users may have different preferences for the behavior of a mix of applications. For any given mix of applications, there are often a wide range of possible behaviors. In this context, an operating system cannot be expected to determine a priori a single set of application behaviors that works best for all users. There may be many application behaviors that work well for different users under different circumstances for different applications.

It is not the job of the operating system to tell users what is the best way for their applications to behave. We believe that users know best how they want their applications to behave. Instead, the operating system should provide mechanisms that allow users to choose from a wide range of selectable behaviors for a mix of applications. If a wide range of behaviors is possible, the operating system can accommodate whatever application behaviors a user desires for a mix of applications.

To provide a wide range of selectable application mix behaviors, the interface between the operating system and applications and users is crucial. After all, the operating system on its own has no way of knowing the behavior desired by applications and users. It also does not know the resource requirements of the applications. Such information is important for making well-informed scheduling decisions, such as when a real-time application needs to run to meet its timing requirements. For users and applications, the interface determines how easy or how hard it is to control the system behavior for a mix of applications. For instance, in the absence of any higher-level programming abstractions for dealing with timing requirements, a recurring development cost is imposed on each programmer who creates a real-time application.

To reduce the burden of real-time programming and provide effective performance for multimedia applications, SMART provides a simple interface for applications and users that allows access to its underlying resource management mechanisms. This interface (1) enables the operating system to manage resources more effectively by using knowledge of application-specific timing requirements, (2) provides dynamic feedback to real-time applications to inform them if their time constraints cannot be met so that they can adapt to the current loading condition, (3) gives end users simple predictable controls that can be used to bias the allocation of resources according to their preferences.

This chapter presents the design of the SMART interface and its usage model, with particular emphasis on the real-time application programming interface. The interface provides two kinds of support for multimedia applications. One is to support the developers of multimedia applications that are faced with writing applications that have dynamic and adaptive real-time requirements. The other is to support the end users of multimedia applications, each of whom may have different preferences for how a given mix of applications should run. The SMART combination of application-level real-time support with predictable controls for expressing user preferences affords a wide range of rich predictable behaviors for mixes of multimedia applications and their users.

## 3.1. Application Developer Support

Multimedia application developers are faced with the problem of writing applications with real-time requirements. They know the time constraints that should be met in these applications and know how to allow these applications to adapt and degrade gracefully when not all time constraints can be met. The problem is that current operating system practice, as typified by UNIX, does not provide an adequate amount of functionality for supporting these applications. For example, in dealing with time in UNIX time-sharing, an application can obtain simple timing information such as elapsed wall clock time and accumulated execution time during its computations. An application can also tell the scheduler to delay the start of a computation by “sleeping” for a duration of time. But it is not possible for an application to ask the scheduler to complete a computation within certain time constraints, nor can it obtain feedback from the scheduler on whether or not it is possible for a computation to complete within the desired time constraints. The application ends up finding out only after the fact that its efforts were wasted on results that could not be delivered on time. The lack of system support exacerbates the difficulty of writing applications with real-time requirements and results in poor application performance.

To address these limitations, SMART provides to the application developer three kinds of programming constructs: a *time constraint* to allow an application to express to the scheduler the timing requirements of a given block of application code, a *notification* to allow the scheduler to inform the application via an upcall when its timing requirements cannot be met, and an *availability* to indicate the availability of processing time. In particular, applications can have blocks of code that have time constraints and blocks of code that do not, thereby allowing application developers to freely mix real-time and non-real-time computations. The SMART application programming constructs are described in further detail in Section 3.4.

By allowing applications to inform the scheduler of their time constraints, the scheduler can optimize how it sequences the resource requests of different applications to meet as many time constraints as possible. It can delay those computations with less stringent timing requirements to allow those with more stringent requirements to execute. It can use this knowledge of the timing requirements of all applications to estimate the load on the system and determine which time constraints can and cannot be met. By providing notifications, the scheduler frees applications from the burden of second guessing the system to determine if their time constraints can be met. By having the scheduler provide information on the availability of resources to applications, an adaptive real-time application can determine how best to adjust its execution rate when its timing requirements cannot be met.

The model of interaction provided by SMART is one of propagating information between applications and the scheduler to facilitate their cooperation in managing resources. Neither can do the job effectively on its own. Only the scheduler can take responsibility for arbitrating resources among competing applications, but it needs applications to inform it of their requirements to do that job effectively. Different applications have different adaptation policies, but they need support from the scheduler to estimate the load and determine when and what time constraints cannot be met.

Note that time constraints, notifications, and availabilities are intended to be used by application writers to support their development of real-time applications; the end user of such applications need not know anything about these constructs or anything about the timing requirements of the applications.

## 3.2. End User Support

Different users may have different preferences for how processing time should be allocated among a set of applications. Not all applications are always of equal importance to a user. For example, a user may want to ensure that an important video teleconference be played at the highest image and sound quality possible, at the sacrifice if need be of the quality of a television program that the user was just watching to pass the time. However, current practice, as typified by UNIX, provides little in the way of predictable controls to bias the allocation of resources in accordance with user preferences. For instance, in UNIX time-sharing, all that a user is given is a “nice” knob [1] whose setting is poorly correlated to user observable behavior [50].

SMART provides two parameters to predictably control processor allocation: *priority* and *share*. These parameters can be used to bias the allocation of resources to provide the best performance for those applications which are more important to the user.

The user can specify that applications have different priorities. The application with the higher priority is favored whenever there is contention for resources. The system will not degrade the performance of a higher priority application to execute a lower priority application. For instance, suppose we have two real-time applications, one with higher priority than the other, and the lower priority application having a computation with a more stringent time constraint. If the lower priority application needs to execute first in order to meet its time constraint, the system will allow it to do so as long as its execution does not cause the higher priority application to miss its time constraint. Among applications with the same priority, the user can specify the share of each application. This will allow each application to receive an allocation of resources in proportion to its respective share whenever there is contention for resources.

Our expectation is that most users will run the applications in the default priority level with equal shares. This is the system default and requires no user parameters. The user may wish to adjust the proportion of shares between the applications occasionally. A simple graphical interface can be provided to make the adjustment as simple and intuitive as adjusting the volume of a television or the balance of a stereo output. The user may want to use the priority to handle specific circumstances. Suppose we wish to ensure that an audio telephony application always can execute; this can be achieved by running the application with high priority.

## 3.3. SMART Use of Information

Fundamental to the design of SMART is the separation of importance information as expressed by user preferences from the urgency information as expressed by the time constraints of the applications. Prematurely collapsing urgency and importance information into a single priority value, as is the case with standard UNIX SVR4 real-time scheduling, results in a significant loss of information and denies the scheduler the necessary knowledge to perform its job effectively. By providing both dimensions of information, the scheduler can do a better job of sequencing the resource requests in meeting the time constraints, while ensuring that even if not all time constraints can be met, the more important applications will at least meet their time constraints.

While SMART accounts for both application and user information in managing resources, it in no way imposes draconian demands on either application developers or end users for information they cannot or choose not the provide. The design provides reasonable default behavior as well as incrementally better results for incrementally more information. By default, an end user can just run an application as he would today and obtain fair behavior. If he desires that more resources should be allocated to a given application, SMART provides simple controls that can be used to express that to the scheduler. Similarly, an application developer need not use any of SMART’s real-time programming constructs unless he desires such functionality. Alternatively, he might choose to use only time constraints, in which case he need not know about notifications or availabilities. When the functionality is not needed, the information need not be provided. However, unlike other systems, when the real-time programming support is desired, as is often the case with multimedia applications, SMART has the ability to provide it.

## 3.4. SMART Real-time API

Having described the basic usage model for SMART and presented an overview of the SMART interface, we now provide a more detailed description of the real-time application programming constructs and their use. An example that shows how these constructs are used in a real-time video application is described in Section 3.5.

### 3.4.1. Time Constraint

The time constraint is used to allow an application to inform the scheduler of the real-time characteristics of a computation, as defined by a block of application code. A time constraint consists of two parameters:

- *deadline*: The deadline is the time by which the application requests that the block of code be completed.

- *cpu-estimate*: The cpu-estimate is an estimate of the amount of processing time required for the block of code.

Because the cpu-estimate is an estimate of the amount of processing time required to process a time constraint, it may differ from the actual amount of time required. The cpu-estimate may be greater than or less than the actual processing time required. We define how SMART behaves in each of these cases. SMART uses the cpu-estimates in conjunction with the deadlines to determine if an activity will be able to meet its time constraint given the current system load. If the cpu-estimate of an activity is larger than the actual processing time required, SMART will end up being more conservative in deciding whether the activity can meet its deadline. If the cpu-estimate of an activity is smaller than the actual processing time required, SMART may not allocate enough processing time for the activity to complete its time constraint.

Note that SMART leaves the decision up to the programmer regarding how aggressive or how conservative to be with the use of the cpu-estimate. This is a programmer-dependent and application-dependent decision. It is not the job of the operating system to dictate how this decision is made. This is something that a programmer should know best and is not something that can be automated. As a result, SMART provides the programmer with the ability to provide any cpu-estimate and thereby choose from a wide range of possible application mix behaviors. To provide a better understanding of how these cpu-estimates can be determined, we discuss one estimation method in Section 3.5.

SMART treats the cpu-estimate as only an estimate of processing time. An activity is not necessarily limited to only using as much processing time as is indicated by its cpu-estimate. If an activity needs more processing time than indicated by its cpu-estimate, the activity can obtain additional processing time in accordance with its priority and share. The activity will be allowed to run for more than its cpu-estimate only if doing so does not conflict with the resource requirements of any more important activities, where importance is determined based on priorities and shares. For instance, if a real-time activity is the highest priority activity, that activity will be given any extra processing time it requires. However, if a real-time activity is a low priority activity and the system is overloaded, the activity may not be given the extra processing time required to meet its deadline because doing so would cause other higher priority activities to miss their deadlines. The use of priorities and shares for determining if an activity can exceed its cpu-estimate is consistent with their use in determining if an activity can meet its time constraint when the system is overloaded.

If the system is overloaded, there will typically be some low importance real-time activities that are not able to meet their time constraints. A question arises as to what is done with such activities. One possibility is for the operating system to abort the execution of the block of code associated with the given time constraint. This would require applications to be written in such a way to deal with such abort events, which would make many real-time programs more difficult to write. Another possibility is to treat a real-time activity as a conventional activity once its time constraint has been missed, but we have already pointed out that real-time and conventional activities typically have very different requirements that need to be met. SMART allows real-time activities that are not able to meet their time constraints to run for some portion of their requested processing time before their deadline. The portion of the requested processing time that is given is assigned in accordance with an activity's priority and share, such that the execution of the given activity does not conflict with the requirements of any more important activities. This processing time may be used by the application in whatever manner the application deems appropriate for dealing with a missed time constraint, such as handling a notification or gracefully adapting and degrading by setting a more relaxed time constraint for its computation. Note that when the system is heavily overloaded, the less important activities may not receive any processing time before their respective deadlines. Once the deadline of a real-time activity has passed, the real-time activity is allowed to execute in accordance with its priority and share, much in the same way as an activity that has exceeded its cpu-estimate.

By default, if the deadline is not specified, the time constraint is simply ignored. By default, if the cpu-estimate is not specified, the system conservatively assumes that the application requires whatever processing time is available until the deadline.

### 3.4.2. Notification

The notification is used to allow an application to request that the scheduler inform it whenever its deadline cannot be met. A notification consists of two parameters:

- *notify-time*: The notify-time is the time after which the scheduler should inform the respective application if it is unlikely to complete its computation before its deadline.

- *notify-handler*: The notify-handler is a function that the application registers with the scheduler. It is invoked via an upcall mechanism from the scheduler when the scheduler notifies the application that its deadline cannot be met.

The notify-time is used by the application to control when the notification upcall is delivered. For instance, if the notify-time is set equal to zero, then the application will be notified immediately if early estimates by the scheduler indicate that its deadline will not be met. On the other hand, if the notify-time is set equal to the deadline, then the application will not be notified until after the deadline has passed if its deadline was not met.

The combination of the notification upcall with the notify-handler frees applications from the burden of second guessing the system to determine if their time constraints can be met, and allows applications to choose their own policies for deciding what to do when a deadline is missed. For example, upon notification, the application may choose to discard the current computation, perform only a portion of the computation, or perhaps change the time constraints. This feedback from the system enables adaptive real-time applications to degrade gracefully.

The processing of a notification by an application requires some amount of processing time. A question that arises is when should a notification be processed and how much processing time should an application be allowed to use in processing a notification? One solution would be to treat notifications as high priority events which are processed immediately. A problem with this approach is that a low priority application could take over the system by doing all its required computing in a notification handler. The system could limit the amount of processing that can occur in a notification handler and preempt the application once that limit is reached. However, the choice of such a limit could arbitrarily limit the functionality of some applications which really needed the extra processing time for their notification handlers. Moreover, high priority applications could still be adversely impacted by notification processing by low priority applications if there are many low priority applications that need to be notified. In SMART, a notification is instead processed as part of the normal allocation of processing time that is given to an application. As a result, a high priority application will have its notification processed at a high priority and a low priority application will have its notification processed at a low priority. In this way, notification processing by lower priority applications will not disrupt the execution of higher priority applications.

By default, if the notify-time is not specified, the application is not notified if its deadline cannot be met. In addition, if no notify-handler is registered, the notify-time is ignored.

### 3.4.3. Availability

When it is not possible to meet the time constraints of an application due to the loading condition of the system, the application may adapt to the loading condition by reducing the quality of its results to reduce its resource consumption. When the load on the system eventually reduces, the application would like to return to providing a higher quality of service. To enable applications to obtain this kind of system load information, the scheduler provides availabilities to applications at their request. An availability is an estimate of the processor time consumption of an application relative to its processor time allocation. It consists of two parameters:

- *consumption-rate*: The consumption-rate is the percentage of the processor that is being consumed by the application.
- *allocation-rate*: The allocation-rate is the percentage of the processor that the application can use as determined by the scheduler based on the priority and share of the application.

If the allocation-rate is larger than the consumption-rate, the application is using less than its allocation of the processor. If the allocation-rate is less than the consumption-rate, the application is using more than its allocation of the processor. For example, suppose we have two applications with equal priority and equal share, one of which only needs 25% of the processor while the other one needs 55% of the processor. Then the respective (consumption-rate, allocation-rate) of each application would be (25, 50) and (55, 50), respectively. By comparing its consumption-rate with its allocation-rate, an application can determine if it can consume a larger portion of processing time and thereby deliver a higher quality of service.

## 3.5. Programming Example

To illustrate how the SMART application interface makes it easier to develop a real-time application, we describe the development of an application which plays multimedia audio and video. The application described is the Integrated Media Streams (IMS) Player from Sun Microsystems Laboratories. This application displays synchronized audio and video streams from local storage. Each media stream flows under the direction of an independent thread of control, which we refer to as a media player. The audio and video players communicate through a shared memory region and use timestamps to synchronize the display of the media streams. The application adapts to its system environment by adjusting the quality of playback based on the system load.

We note that the timing and processing requirements of the media player are not strictly periodic in nature. Most of the timestamps of the media streams occur in a cyclic fashion, but some media samples are spaced further apart in time because intermediate samples were lost when the media streams were first recorded; the media player was designed to be able to accurately playback media streams even if they have aperiodic timestamps. In addition, the video player processes JPEG compressed video, which results in a significant amount of variation in frame processing times.

The application was developed and tuned for the UNIX SVR4 time-sharing scheduler in the Solaris operating system. We describe what it took to develop the IMS Player for UNIX SVR4, then discuss how we modified it for SMART.

### 3.5.1. Video Player

The video player reads a timestamped JPEG video input stream from local storage, uncompresses it, dithers it to 8-bit pseudo-color, and renders it directly to the frame buffer. When the video player is not used in synchrony with an audio player, the player uses the timestamps on the video input stream to determine when to display each frame and whether a given frame is early or late. When used in conjunction with the audio player, the video player attempts to synchronize its output with that of the audio device by matching the timestamps of the video frames with the timestamps of the audio samples being played. In particular, since humans are more sensitive to intra-stream audio asynchronies (i.e. audio delays and drop-outs) than to asynchronies involving video, the thread controlling the audio stream free-runs as the master time reference and the video "slave" thread uses the information the audio player posts into the shared memory region to determine when to display its frames.

If the video player is ready to display its frame early, it delays the display of the frame until the appropriate time; but if it is late, it discards its current frame on the assumption that continued processing will cause further delays later in the stream. The application defines early and late as more than 20 ms early or late with respect to the audio. For UNIX SVR4, the video player must determine entirely on its own whether or not each video frame can be displayed on time. This is done by measuring the amount of wall clock time that elapses during the processing of each video frame. An exponential average [20] of the elapsed wall clock time of previously displayed frames is then used as an estimate for how long it will take to process the current frame. If the estimate indicates that the frame will complete too early (more than 20 ms early), the video player sleeps an amount of time necessary to delay processing to allow the frame to be completed at the right time. If the estimate indicates that the frame will be completed too late (more than 20 ms late), the frame is discarded.

The application adapted to run on SMART uses the same mechanism as the original to delay the frames that would otherwise be completed too early. We simply replace the application's discard mechanism with a time constraint system call to inform SMART of the time constraints for a given block of application code, along with a signal handler to process notifications of time constraints that cannot be met. The time constraint informs SMART of the deadline for the execution of the block of code that processes the video frame. The deadline is set to the time the frame is considered late, which is 20 ms after the ideal display time. It also provides an estimate of the amount of execution time for the code calculated in a similar manner as the original program. In particular, an exponential average of the execution times of previously displayed frames scaled by 10% is used as the estimate. Upon setting the given time constraint, the application requests that SMART provide a notification to the application right away if early estimates predict that the time constraint cannot be met. When a notification is sent to the application, the application signal handler simply records the fact that the notification has been received. If the notification is received by the time the application begins the computation to process and display the respective video frame, the frame is discarded; otherwise, the application simply allows the frame be displayed late.

Figures 3-1 and 3-2 indicate that simple exponential averaging based on previous frame execution times can be used to provide reasonable estimates of frame execution times even for JPEG compressed video in which frame times vary from one frame to another. The data in Figure 3-1 was for a network news programming video sequence while the data in Figure

3-2 was for a television entertainment video sequence. The data in each figure was collected over a 300 second time interval of the respective video sequence. Each figure shows the actual execution time for each video frame and the estimate error, which is the difference between the estimated and actual execution time for each frame. Note that the slight positive bias in the estimate error is due to the 10% scaling in the estimate versus the actual execution time.

As expected, the execution time for processing JPEG video frames varies considerably over the course of the video sequence. Different frames take different amounts of time to decompress, depending on the amount of scene complexity in the respective frame. However, the processing time required by successive video frames in the sequence is often similar because typically not that much content changes from one video frame to the next. Exponential averaging is able to take advantage of this characteristic to do a reasonable job of predicting the execution time of a given frame by using the execution times of immediately preceding video frames. At the same time, since the average is dynamically adjusted and gives most weight to recent frame execution times, it is able to provide reasonable estimates even with substantial variation in frame execution times over the course of a video sequence. The results also illustrate the difficulty of using a more static resource reservation scheme based on a single processing time estimate for all video frames. Using the upper bound on the processing time as an estimate may yield a low utilization of resources; using the average processing time may cause too many deadlines to be missed.

There are cases when the scene in a video sequence may change quite suddenly between two successive frames. For instance, an action-packed entertainment programming sequence would typically have a number of sudden scene changes. In these cases, the processing time can vary considerably between two successive frames. As a result, the estimate error from exponential averaging will be higher during such scene changes. In fact, the estimate error shown in Figure 3-2 which is from entertainment programming is higher than the estimate error shown in Figure 3-1 which is from network news programming. The reason is that there are many more sudden scene changes in the entertainment programming. In contrast, the network news programming has more talking head video clips which vary little from frame to frame.

While it is more difficult to provide an accurate estimate of video frame processing time during sudden scene changes from one frame to the next, it is also more difficult for users to see many of the details in the video during such rapid scene changes. As a result, if larger estimate errors during sudden scene changes caused the corresponding video frames to be scheduled less accurately, the visual effect of the inaccuracy is often not very noticeable to the user.

While we have used JPEG video for these measurements, reasonable estimates of frame execution times can also be provided for other video formats such as MPEG video [5]. In the case of MPEG video, averaging would be required for each type of frame. By allowing time constraints to be specified on a frame-by-frame basis, SMART provides enough flexibility to handle real-time requirements that may be quite dynamic and aperiodic in nature.

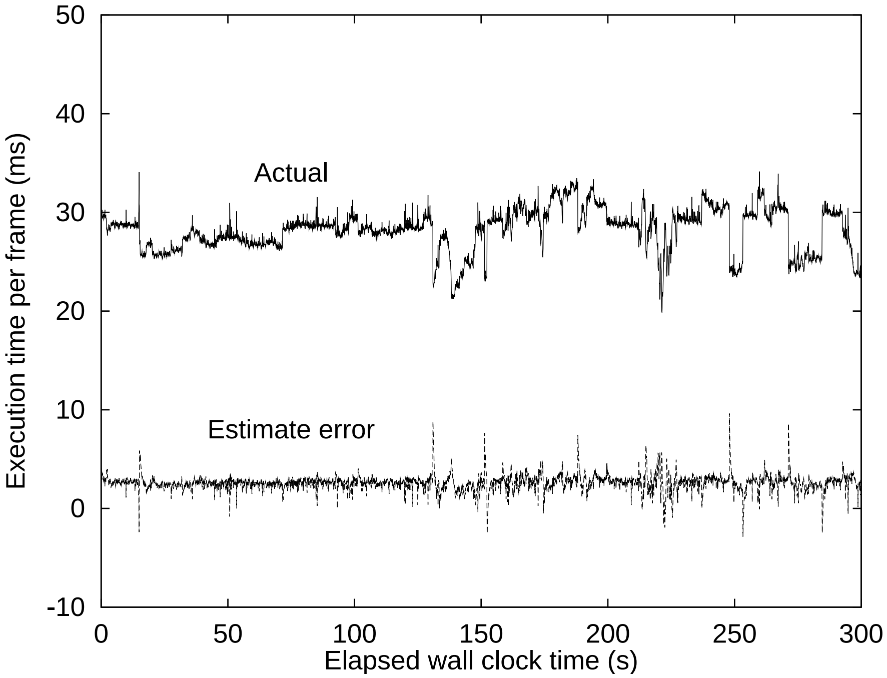


Figure 3-1. Actual vs. estimated frame execution times for JPEG network news video

### 3.5.2. Audio Player

The audio player reads a timestamped 8-bit μ-law audio input stream from local storage and outputs the audio samples to the audio device. The processing of the 8-bit μ-law monaural samples is done in 512 byte segments. To avoid audio dropouts, the audio player takes advantage of buffering available on the audio device to work ahead in the audio stream when processor cycles are available. Up to 1 second of workahead is allowed. For each block of code that processes an audio segment, the audio player aims to complete the segment before the audio device runs out of audio samples to display. The deadline communicated to SMART is therefore set to the display time of the last audio sample in the buffer. The estimate of the execution time is again computed by using an exponential average of the measured execution times for processing previous audio segments. It turns out that the processing time for all the audio segments is relatively constant, so that there is very little error in the execution time estimates. Audio segments that cannot be processed before their deadlines are simply displayed late. Note that because of the workahead feature and the audio device buffering, the resulting deadlines can be highly aperiodic.

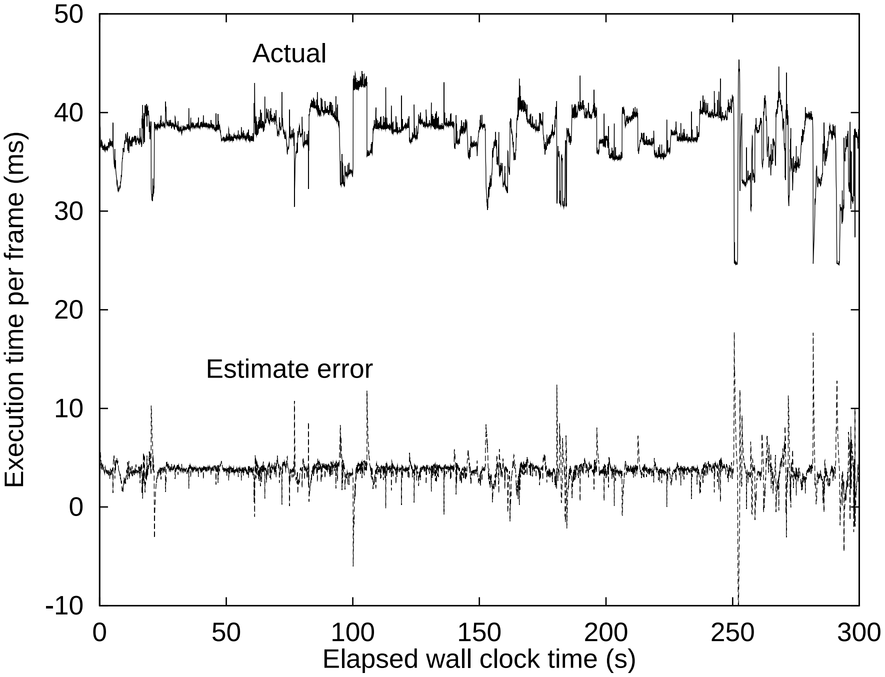


Figure 3-2. Actual vs. estimated frame execution times for JPEG entertainment video

We see from this example that the SMART interface can be used to reduce the burden of developing real-time applications. It facilitates the communication of application timing requirements between the application and the operating system. Because the interface allows time constraints to be specified on a per instance basis, it can be used to support real-time applications that are highly adaptive, dynamic, and aperiodic in nature. As we show in Chapter 6, when coupled with an underlying scheduling algorithm, the SMART interface helps to provide significant improvements in the performance of multimedia applications over other scheduling approaches.

## 3.6. Summary

We have described a new scheduling interface that provides effective support for multimedia applications in a general-purpose computing environment. The SMART interface accounts for user preferences and allows applications to cooperate with the scheduler in supporting their real-time requirements. In particular, the interface allows users to prioritize and proportionally share resources among applications according to their preferences. Furthermore, the scheduler cooperates with applications so that the scheduler can employ application-specific timing information it needs to manage resources effectively, and real-time applications can obtain the necessary dynamic feedback to enable them to adapt to changes in the system load to provide the best possible quality of service.

# 4 The SMART Scheduling Algorithm

A well-designed interface between the operating system and its applications and users provides users with the ability to select from a wide range of possible behaviors for a given mix of applications. However, an interface alone is ineffective without an underlying scheduling algorithm that can actually make good use of hardware resources in supporting such possible application behaviors.

In this chapter, we describe the underlying scheduling algorithm in SMART. This algorithm effectively supports a wide range of application behaviors, as expressed using the SMART interface. We first describe the principles of operations used in the design of the SMART scheduler. We then give an overview of the rationale behind the design, followed by an overview of the algorithm and then the details.

## 4.1. Principles of Operations

It is the scheduler's objective to deliver the behavior expected by the user in a manner that maximizes the overall value of the system to its users. We have reduced this objective to the following six principles of operations:

- *Priority.* The system should not degrade the performance of a high priority application in the presence of a low priority application.

- *Proportional sharing among real-time and conventional applications in the same priority class.* Proportional sharing applies only if the scheduler cannot satisfy all the requests in the system. The system will fully satisfy the requests of all applications requesting less than their proportional share. The resources left over after satisfying these requests are distributed proportionally among activities that can use the excess.

While it is relatively easy to control the execution rate of conventional applications, the execution rate of a real-time application is controlled by selectively shedding computations in as even a rate as possible.

- *Graceful transitions between fluctuations in load.* The system load varies dynamically, new applications come and go, and the resource demand of each application may also fluctuate. The system must be able to adapt to the changes gracefully.

- *Satisfying real-time constraints and fast interactive response time in underload.* If real-time and interactive activities request less than their proportional share, their time constraints should be honored when possible, and the interactive response time should be short.

- *Trading off instantaneous fairness for better real-time and interactive response time.* While it is necessary that the allocation is fair on average, insisting on being fair instantaneously at all times would cause many more deadlines to be missed and deliver poor response time to short running activities. We will tolerate some instantaneous unfairness so long as the extent of the unfairness is bounded. This is the same motivation behind the design of multi-level feedback schedulers [39] to improve the response time of interactive activities.

- *Notification of resource availability.* SMART allows applications to specify if and when they wish to be notified if it is unlikely that their computations will be able to complete before their given deadlines.

## 4.2. Rationale and Overview

Real-time and conventional applications have very diverse characteristics. Real-time applications have some well-defined computation which must be completed before an associated deadline. The goal of a real-time application is typically to complete as many computations before their respective deadlines as possible. In contrast, conventional applications have no explicit deadlines and their computations are often harder to predict. Instead, the goal is typically to deliver good response time for interactive applications and fast program completion time for batch applications. These characteristics are summarized in Table 4-1. It is this diversity that makes devising an integrated scheduling algorithm difficult. A real-time scheduler uses real-time constraints to determine the execution order, but conventional activities do not have real-time constraints. Adding periodic deadlines to conventional activities is a tempting design choice, but it introduces artificial constraints that reduce the effectiveness of the system. On the other hand, a conventional activity scheduler has no notion of real-time constraints; the notion of time-slicing the applications to optimize system throughput does not serve real-time applications well.

| | **Real-Time Applications** | **Conventional Applications** | |
|---|---|---|---|
| | | **Interactive** | **Batch** |
| **Deadlines** | Yes | No | No |
| **Quantum of Execution** | Service time: no value if the entire activity is not executed | Arbitrary choice | Arbitrary choice |
| **Resource Requirement** | A slack is usually present | Relinquishes machine while waiting for human response | Can consume all processor cycles until it completes |
| **Quality of Service Metric** | Number of deadlines met | Response time | Program completion time |

Table 4-1. Categories of applications

The crux of the solution is not to confuse *urgency* with *importance*. An urgent activity is one which has an immediate real-time constraint. An important activity is one with a high priority, or one that has been the least serviced proportionally among applications with the same priority. An urgent activity may not be the one to execute if it requests more resources than its fair share. Conversely, an important activity need not be run immediately. For example, a real-time activity that has a higher priority but a later deadline may be able to tolerate the execution of a lower priority activity with an earlier deadline. Our algorithm separates the processor scheduling decisions into two steps; the first identifies all the candidates that are considered important enough to execute, and the second chooses the activity to execute based on urgency considerations.

A key characteristic of this two-step scheduling algorithm is that it avoids the tyranny of the urgent. That is, there are often many urgent activities that need to get done, but not enough time to do all of them completely within their time constraints. However, trying to focus on getting the urgent activities done while neglecting the less time constrained but more important activities that need to get done is typically a path to long term disaster. Instead, what our algorithm effectively does is it gets things that are more urgent done sooner, but defers less important activities as needed to ensure that the more important activities can meet their requirements. In particular, what candidates are considered important enough to execute depends on the load on the system. If the system is lightly loaded such that all activities can run and meet their requirements, then all activities will be considered important enough to execute. The algorithm will then order all activities based on urgency to do the best job of meeting the deadlines of all real-time activities. If the system is heavily loaded such that not all activities can run and meet their requirements, then the algorithm will only consider as candidates the most important activities that can meet their requirements with the available processing time.

While urgency is specific to real-time applications, importance is common to all the applications. We measure the importance of an application by a *value-tuple*, which is a tuple with two components: priority and the *biased virtual finishing time* (*BVFT*). Priority is a static quantity either supplied by the user or assigned the default value; BVFT is a dynamic quantity the system uses to measure the degree to which each activity has been allotted its proportional share of resources. The formal definition of the BVFT is given in Section 4.3. We say that activity *A* has a higher value-tuple than activity *B* if *A* has a higher static priority or if both *A* and *B* have the same priority and *A* has an earlier BVFT. The value-tuple effectively provides a way to express what would otherwise be a non-obvious utility function for capturing both the notions of prioritized and proportional sharing.

The SMART scheduling algorithm used to determine the next activity to run is as follows:

1. If the activity with the highest value-tuple is a conventional activity (an activity without a deadline), schedule that activity.

2. Otherwise, create a candidate set consisting of all real-time activities with higher value-tuple than that of the highest value-tuple conventional activity. (If no conventional activities are present, all the real-time activities are placed in the candidate set.)

3. Apply the best-effort real-time scheduling algorithm [46] on the candidate set, using the value-tuple as the priority in the original algorithm. By using the given deadlines and service-time estimates, find the activity with the earliest deadline whose

execution does not cause any activities with higher value-tuples to miss their deadlines. This is achieved by considering each candidate in turn, starting with the one with the highest value-tuple. The algorithm attempts to schedule the candidate into a working schedule which is initially empty. The candidate is inserted in deadline order in this schedule provided its execution does not cause any of the activities in the schedule to miss its deadline. The scheduler simply picks the activity with the earliest deadline in the working schedule.

4. If an activity cannot complete its computation before its deadline, send a notification to inform the respective application that its deadline cannot be met.

The following sections provide a more detailed description of the BVFT, and the best-effort real-time scheduling technique.

## 4.3. Biased Virtual Finishing Time

The notion of a *virtual finishing time (VFT)*, which measures the degree to which the activity has been allotted its proportional share of resources, has been previously used in describing fair queueing algorithms [6, 12, 57, 65, 69]. These proportional sharing algorithms associate a VFT with each activity as a way to measure the degree to which an activity has received its proportional allocation of resources. We augment this basic notion of a VFT in the following ways. First, our use of virtual finishing times incorporates activities with different priorities. Second, we add to the virtual finishing time a bias, which is a bounded offset used to measure the ability of conventional activities to tolerate longer and more varied service delays. The biased virtual finishing time allows us to provide better interactive and real-time response without compromising fairness. Finally and most importantly, fair queueing algorithms such as weighted fair queueing (WFQ) execute the activity with the earliest virtual finishing time to provide proportional sharing. SMART only uses the biased virtual finishing time in the selection of the candidates for scheduling, and real-time constraints are also considered in the choice of the application to run. This modification enables SMART to handle applications with aperiodic constraints and overloaded conditions.

Our algorithm organizes all the activities into queues, one for each priority. The activities in each queue are ordered in increasing BVFT values. Each activity has a *virtual time* which advances at a rate proportional to the amount of processing time it consumes divided by its share. Suppose the current activity being executed has share $S$ and was initiated at time $\tau$. Let $v(\tau)$ denote the activity's virtual time at time $\tau$. Then the virtual time $v(t)$ of the activity at current time $t$ is

$$v(t) = v(\tau) + \frac{t - \tau}{S}. \tag{1}$$

Correspondingly, each queue has a *queue virtual time* which advances only if any of its member activities is executing. The rate of advance is proportional to the amount of processing time spent on the activity divided by total number of shares of all activities on the queue. To be more precise, suppose the current activity being executed has priority $P$ and was initiated at time $\tau$. Let $V_P(\tau)$ denote the queue virtual time of the queue with priority $P$ at time $\tau$. Then the queue virtual time $V_P(t)$ of the queue with priority $P$ at current time $t$ is

$$V_P(t) = V_P(\tau) + \frac{t - \tau}{\sum_{a \in A_P} S_a}, \tag{2}$$

where $S_a$ represents the share of application $a$, and $A_P$ is the set of applications with priority $P$.

Previous work in the domain of packet switching provides a theoretical basis for using the difference between the virtual time of an activity and the queue virtual time as a measure of whether the respective activity has consumed its proportional allocation of resources [12, 57]. If an activity's virtual time is equal to the queue virtual time, it is considered to have received its proportional allocation of resources. An earlier virtual time indicates that the activity has less than its proportional share, and, similarly, a later virtual time indicates that it has more than its proportional share. Since the queue virtual time advances at the same rate for all activities on the queue, the relative magnitudes of the virtual times provide a relative measure of the degree to which each activity has received its proportional share of resources.

The virtual finishing time refers to the virtual time of the application, had the application been given the currently requested quantum. The quantum for a conventional activity is the unit of time the scheduler gives to the activity to run before being rescheduled. The quantum for a real-time activity is the application-supplied estimate of its service time. A useful property of the virtual finishing time, which is not shared by the virtual time, is that it does not change as an activity executes and uses up its time quantum, but only changes when the activity is rescheduled with a new time quantum.

In the following, we step through all the events that lead to the adjustment of the biased virtual finishing time of an activity. Let the activity in question have priority $P$ and share $S$. Let $\beta(t)$ denote the BVFT of the activity at time $t$.

*activity creation time.* When an activity is created at time $\tau_0$, it acquires as its virtual time the queue virtual time of the its corresponding queue. Suppose the activity has time quantum $Q$, then its BVFT is

$$\beta(\tau_0) = V_P(\tau_0) + \frac{Q}{S}. \qquad (3)$$

*Completing a Quantum.* Once an activity is created, its BVFT is updated as follows. When an activity finishes executing for its time quantum, it is assigned a new time quantum $Q$. As a conventional activity accumulates execution time, a bias is added to its BVFT when it gets a new quantum. That is, let $b$ represent the increased bias and $\tau$ be the time an activity's BVFT was last changed. Then, the activity's BVFT is

$$\beta(t) = \beta(\tau) + \frac{Q}{S} + \frac{b}{S}. \qquad (4)$$

The bias is used to defer long running batch computations during transient loads to allow real-time and interactive activities to obtain better immediate response time. The bias is increased in a manner similar to the way priorities and time quanta are adjusted in UNIX SVR4 to implement time-sharing [68]. The total bias added to an application's BVFT is bounded. Thus, the bias does not change either the rate at which the BVFT is advanced or the overall proportional allocation of resources. It only affects the instantaneous proportional allocation. User interaction causes the bias to be reset to its initial value. Real-time activities have zero bias.

The idea of a dynamically adjusted bias based on execution time is somewhat analogous to the idea of a decaying priority based on execution time which is used in multilevel-feedback schedulers. However, while multilevel-feedback affects the actual average amount of resources allocated to each activity, bias only affects the response time of an activity and does not affect its overall ability to obtain its proportional share of resources. By combining virtual finishing times with bias, the BVFT can be used to provide both proportional sharing and better system responsiveness in a systematic fashion.

*Blocking for I/O or events.* A blocked activity should not be allowed to accumulate credit to a fair share indefinitely while it is sleeping; however, it is fair and desirable to give the activity a limited amount of credit for not using the processor cycles and to improve the responsiveness of these activities. Therefore, SMART allows the activity to remain on its given priority queue for a limited duration which is equal to the lesser of the deadline of the activity (if one exists), or a system default. At the end of this duration, a sleeping activity must leave the queue, and SMART records the difference between the activity's and the queue's virtual time. This difference is then restored when the activity rejoins the queue once it becomes runnable. Let $E$ be the execution time the activity has already received toward completing its time quantum $Q$, $B$ be its current bias, and $v(t)$ denote the activity's virtual time. Then, the difference $\Delta$ is

$$\Delta = v(t) - V_P(t), \tag{5}$$

where

$$v(t) = \beta(t) - \frac{Q-E}{S} - \frac{B}{S}. \tag{6}$$

Upon rejoining the queue, its bias is reset to zero and the BVFT is

$$\beta(t) = V_P(t) + \Delta + \frac{Q}{S}. \tag{7}$$

*Reassigned user parameters.* If an activity is given a new priority, it is reassigned to the queue corresponding to its new priority, and its BVFT is simply calculated as in Equation (3). If the activity is given a new share, the BVFT is calculated by having the activity leave the queue with the old parameters used in Equation (6) to calculate Δ, and then join the queue again with the new parameters used in Equation (7) to calculate its BVFT.

## 4.4. Best-effort Real-time scheduling

SMART iteratively selects activities from the candidate set in decreasing value-tuple order and inserts them into an initially empty working schedule in increasing deadline order. The working schedule defines an execution order for servicing the real-time resource requests. It is said to be *feasible* if the set of activity resource requirements in the working schedule, when serviced in the order defined by the working schedule, can be completed before their respective deadlines. It should be noted that the resource requirement of a periodic real-time activity includes an estimate of the processing time required for its future resource requests.

To determine if a working schedule is feasible, let $Q_j$ be the processing time required by activity $j$ to meet its deadline, and let $E_j$ be the execution time activity $j$ has already spent running toward meeting its deadline. Let $F_j$ be the fraction of the processor required by a periodic real-time activity; $F_j$ is simply the ratio of an activity's service time to its period if it is a periodic real-time activity, and zero otherwise. Let $D_j$ be the deadline of the activity. Then, the estimated resource requirement of activity $j$ at a time $t$ such that $t \geq D_j$ is:

$$R_j(t) = Q_j - E_j + F_j \times (t - D_j), t \geq D_j . \tag{8}$$

A working schedule $W$ is then feasible if for each activity $i$ in the schedule with deadline $D_i$, the following inequality holds:

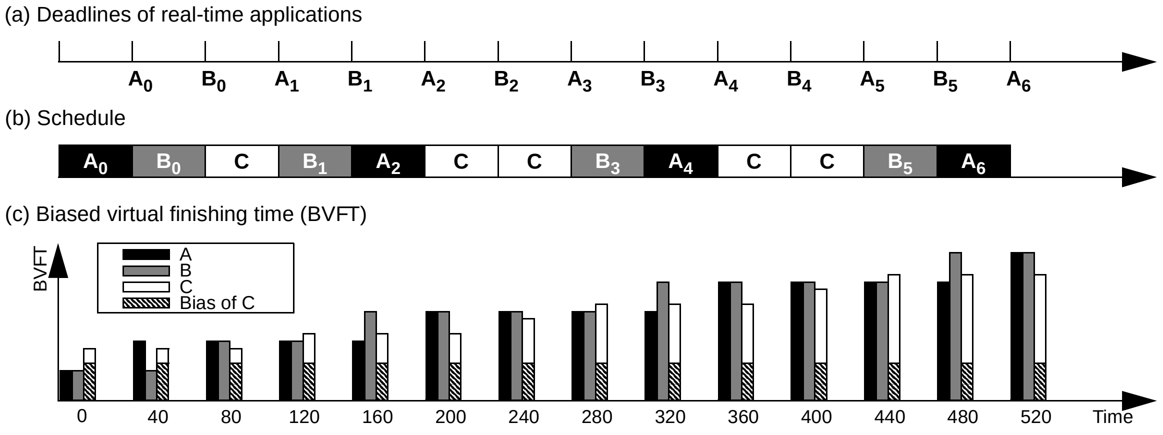
$$D_i \geq t + \sum_{j \in W | D_j \leq D_i} R_j(D_i), \dot{\forall} i \in W . \tag{9}$$

On each activity insertion into the working schedule, the resulting working schedule that includes the newly inserted activity is tested for feasibility. If the resulting working schedule is feasible and the newly inserted activity is a periodic real-time activity, its estimate of future processing time requirements is accounted for in subsequent feasibility tests. At the same time, lower value-tuple activities are only inserted into the working schedule if they do not cause any of the current and estimated future resource requests of higher value-tuple activities to miss their deadlines. The iterative selection process is repeated until SMART runs out of activities or until it determines that no further activities can be inserted into the schedule feasibly. Once the iterative selection process has been terminated, SMART then executes the earliest-deadline runnable activity in the schedule.

If there are no runnable conventional activities and there are no runnable real-time activities that can complete before their deadlines, the scheduler runs the highest value-tuple runnable real-time activity, even though it cannot complete before its deadline. The rationale for this is that it is better to use the processor cycles than allow the processor to be idle. The algorithm is therefore work conserving, meaning that the resources are never left idle if there is a runnable activity, even if it cannot satisfy its deadline.

## 4.5. Example

We now present a simple example to illustrate how the SMART algorithm works. Consider a workload involving two real-time applications, *A* and *B*, and a conventional application *C*. Suppose all the applications belong to the same priority class, and their proportional shares are in the ratio of 1:1:2, respectively. Both real-time applications request 40 ms of computation time every 80 ms, with their deadlines being completely out of phase, as shown in Figure 4-1(a). The applications request to be notified if the deadlines cannot be met; upon notification, the application drops the current computation and proceeds to the computation for the next deadline. The scheduling quantum of the conventional application *C* is also 40 ms and we assume that it has accumulated a bias of 100 ms at this point. Figure 4-1(b) and (c) show the final schedule created by SMART for this scenario, and the BVFT values of the different applications at different time instants.



(a) Deadlines of real-time applications

(b) Schedule

(c) Biased virtual finishing time (BVFT)

Figure 4-1. Example illustrating the behavior of the SMART algorithm

The initial BVFTs of applications *A* and *B* are the same; since *C* has twice as many shares as *A* and *B*, the initial BVFT of *C* is half of the sum of the bias and the quantum length.

Because of the bias, application *C* has a later BVFT and is therefore not run immediately. The candidate set considered for execution consists of both applications, *A* and *B; A* is selected to run because it has an earlier deadline. (In this case, the deadline is used as a tie-breaker between real-time activities with the same BVFT; in general, an activity with an early deadline may get to run over an activity with an earlier BVFT but a later deadline.) When an activity finishes its quantum, its BVFT is incremented. The increment for *C* is half of that for *A* and *B* because the increment is the result of dividing the time quantum by its share. Figure 4-1(c) shows how the activities are scheduled such that their BVFT are kept close together.

This example illustrates several important characteristics of SMART. First, SMART implements proportional sharing properly. In the steady state, *C* is given twice as much resources as *A* or *B,* which reflects the ratio of shares given to the applications. Second, the bias allows better response in temporary overload, but it does not reduce the proportional share given to the biased activity. Because of *C*'s bias, *A* and *B* get to run immediately at the beginning; eventually their BVFTs catch up with the bias, and *C* is given its fair share. Third, the scheduler is able to meet many real-time constraints, while skipping tardy computations. For example, at time 0, SMART schedules application *A* before *B* so as to satisfy both deadlines. On the other hand, at time 120 ms into the execution, realizing that it cannot meet the $A_2$ deadline, it executes application *B* instead and notifies *A* of the missed deadline.

## 4.6. Complexity

The cost of scheduling with SMART consists of the cost of managing the value-tuple list and the cost of managing the working schedule. The cost of managing the value-tuple list in SMART is $O(N)$, where *N* is the number of active activities. This assumes a linear insertion value-tuple list. The complexity can be reduced to $O(\log N)$ using a tree data structure. For small *N,* a simple linear list is likely to be most efficient in practice. The cost of managing the value-tuple list is the same as WFQ.

The worst case complexity of managing the working schedule is $O({N_R}^2)$, where $N_R$ is the number of active real-time activities of higher value than the highest value conventional activity. This worst case occurs if each real-time activity needs to be selected and feasibility tested against all other activities when rebuilding the working schedule. It is unlikely for the worst case to occur in practice for any reasonably large $N_R$. Real-time activities typically have short deadlines so that if there are a large number of real-time activities, the scheduler will determine that there is no more slack in the schedule before all of the activities need to be individually tested for insertion feasibility. The presence of conventional activities in the workstation environment also prevents $N_R$ from growing large. For large $N$, the cost of scheduling with SMART in practice is expected to be similar to WFQ.

A more complicated algorithm can be used to reduce the complexity of managing the working schedule. In this case, a new working schedule can be incrementally built from the existing working schedule as new activities arrive. By using information contained in the existing working schedule, the complexity of building the new working schedule can be reduced to $O(N_R)$. When only deletions are made to the working schedule, the existing working schedule can simply be used, reducing the cost to $O(1)$.

## 4.7. Analysis of the Behavior of the Algorithm

In the following, we describe how the scheduling algorithm follows the principles of operations as laid out in Section 4.1.

### 4.7.1. Priority

Our principle of operation regarding priority is that the performance of high priority activities should not be affected by the presence of low priority activities. As the performance of a conventional activity is determined by its completion time, a high priority conventional activity should be run before any lower priority activity. Step 1 of the algorithm guarantees this behavior because a high priority activity always has a higher value-tuple than any lower priority activity.

On the other hand, the performance metric of a real-time application is the number of deadlines satisfied, not how early the execution takes place. The best-effort scheduling algorithm in Step 3 will run a lower priority activity with an earlier deadline first, only if it can determine that doing so does not cause the high priority activity to miss its deadline. In this way, the system delivers a better overall value to the user. Note that the scheduler uses the timing information supplied by the applications to determine if a higher priority deadline is to be satisfied. It is possible for a higher priority deadline to be missed if its corresponding time estimate is inaccurate.

### 4.7.2. Proportional sharing

Having described how time is apportioned across different priority classes, we now describe how time allocated to each priority class is apportioned between applications in the class. If the system is populated with only conventional activities, we simply divide the cycles in proportion to the shares across the different applications. As noted in Table 4-1, interactive and real-time applications may not use up all the resources that they are entitled to. Any unused cycles are proportionally distributed among those applications that can consume the cycles.

### 4.7.3. Conventional Activities

Let us first consider conventional activities whose virtual finishing time has not been biased. We observe that even though real-time activities may not execute in the order dictated by WFQ, the scheduler will run a real-time activity only if it has an earlier VFT than any of the conventional activities. Thus, by considering all the real-time activities with an earlier VFT as one single application with a correspondingly higher share, we see the SMART treatment of the conventional activities is identical to that of a WFQ algorithm. From the analysis of the WFQ algorithm, it is clear that conventional activities are given their fair shares.

A bias is added to an activity’s VFT only after it has accumulated a significant computation time. As a fixed constant, the bias does not change the relative proportion between the allocation of resources. It only serves to allow a greater variance in instantaneous fairness, thus allowing a better interactive and real-time response in transient overloads.

### 4.7.4. Real-time Activities

We say that a system is *underloaded* if there are sufficient cycles to give a fair share to the conventional activities in the system while satisfying all the real-time constraints. When a system is underloaded, the conventional activities will be serviced often enough with the left-over processor cycles so that they will have later BVFTs than real-time applications. The conventional applications will therefore only run when there are no real-time applications in the system. The real-time activities are thus scheduled with a strict best-effort scheduling algorithm. It has been proven that in underload, the best-effort scheduling algorithm degenerates to an earliest-deadline scheduling algorithm [45], which has been shown to satisfy all scheduling constraints, periodic or aperiodic, optimally [13].

In an underloaded system, the scheduler satisfies all the real-time applications' requests. CPU time is given out according to the amounts requested, which may have a very different proportion from the shares assigned to the applications. The assigned proportional shares are used in the management of real-time applications only if the system is oversubscribed.

A real-time application whose request exceeds its fair share for the current loading condition will eventually accumulate a BVFT later than other applications' BVFTs. Even if it has the earliest deadline, it will not be run immediately if there is a conventional application with a higher value, or if running this application will cause a higher valued real-time application to miss its deadline. If the application accepts notification, the system will inform the application when it determines that the constraint will not be met. This interface allows applications to implement their own degradation policies. For instance, a video application can decide whether to skip the current frame, skip a future frame, or display a lower quality image when the frame cannot be fully processed in a timely fashion. The application adjusts the timing constraint accordingly and informs the system. If the application does not accept notification, however, eventually all the other applications will catch up with their BVFT, and the scheduler will allow the now late application to run.

Just as the use of BVFT regulates the fair allocation of resources for conventional activities, it scales down the real-time activities proportionally. In addition, the bias introduced in the algorithm, as well as the use of a best-effort scheduler among real-time activities with sufficiently high values, allows more real-time constraints to be met.

## 4.8. Comparison with Related Work

Recognizing the need to provide better scheduling to support the needs of modern applications such as multimedia, a number of resource management approaches have been proposed. These approaches can be loosely classified as real-time scheduling, fair queueing, and hierarchical scheduling.

### 4.8.1. Real-time Scheduling

Real-time schedulers such as rate-monotonic scheduling [41, 45] and earliest-deadline scheduling [13, 45] are designed to make better use of hardware resources in meeting real-time requirements. In particular, earliest-deadline scheduling is optimal in underload. However, they do not perform well when the system is overloaded, nor are they designed to support conventional applications.

Resource reservations are commonly combined with real-time scheduling in an attempt to run real-time activities with conventional activities [10, 34, 42, 47]. Reservations are used to allow each application to request a percentage of the processor. These approaches are used with admission control to allow real-time activities to reserve a fixed percentage of the resource in accordance with their resource requirement. Any leftover processing time is allocated to conventional activities using a standard timesharing or round-robin scheduler.

Several differences in these reservation approaches are apparent. While the approaches in [10, 42] take advantage of earliest-deadline scheduling to provide optimal real-time performance in underload, the rate monotonic utilization bound used in [47] and the time interval assignment used in Rialto [34] are not optimal, resulting in lower performance than earliest-deadline approaches. In contrast with SMART, these approaches are more restrictive, especially in the level of control provided for conventional activities. They do not provide a common mechanism for sharing resources across real-time and conventional activities. In particular, with conventional activities being given leftover processing time, their potential starvation is a problem. This problem is exacerbated in Rialto [34] in which even in the absence of reservations, applications with time constraints buried in their source code are given priority over conventional applications [33].

Note that the use of reservations relies on inflexible admission control policies to avoid overload. This is usually done on a first-come-first-serve basis, resulting in later arriving applications being denied resources even if they are more important. To be able to execute later arriving applications, an as yet undetermined higher-level resource planning policy, or worse yet, the user, must renegotiate the resource reservations via what is at best a trial-and-error process.

Unlike reservation mechanisms, best-effort real-time scheduling [46] provides optimal performance in underload while ensuring that activities of higher priority can meet their deadlines in overload. However, it provides no way of scheduling conventional activities and does not support common resource sharing policies such as proportional sharing.

By introducing admission control, SMART can also provide resource reservations with optimal real-time performance. In addition, SMART subsumes best-effort real-time scheduling to provide optimal performance in meeting time constraints in underload even in the absence of reservations. This is especially important for common applications such as MPEG video whose dynamic requirements match poorly with static reservation abstractions [3, 24].

### 4.8.2. Fair Queueing

Fair queueing provides a mechanism which allocates resources to activities in proportion to their shares. It was first proposed for network packet scheduling in [12], with a more extensive analysis provided in [57], and later applied to processor scheduling in [69] as stride scheduling. Recent variants [6, 65] provide more accurate proportional sharing at the expense of additional scheduling overhead. The share used with fair queueing can be assigned in accordance with user desired allocations [69], or it can be assigned based on the activity’s resource requirement to provide resource reservations [57, 66]. When used to provide reservations, an admission control policy is also used.

When shares are assigned based on user desired allocations, fair queueing provides more accurate proportional sharing for conventional activities than previous fair-share schedulers [27, 28]. However, it performs poorly for real-time activities because it does not account for their time constraints. In underload, time constraints are unnecessarily missed. In overload, all activities are proportionally late, potentially missing all time constraints.

When shares are assigned based on activity resource requirements to provide reservations, fair queueing can be effective in underload at meeting real-time requirements that are strictly periodic in their computation and deadline. However, its performance is not optimal in underload and suffers especially in the case of aperiodic real-time requirements. To avoid making all activities proportionally late in overload, admission control is used.

Unlike real-time reservation schedulers, fair queueing can integrate reservation support for real-time activities with proportional sharing for conventional activities [66]. However, shares for real-time applications must then be assigned based on their resource requirements; they cannot be assigned based on user desired allocations.

By providing time constraints and shares, SMART not only subsumes fair queueing, but it can also more effectively meet real-time requirements, with or without reservations. Unlike fair queueing, it can provide optimal real-time performance while allowing proportional sharing based on user desired allocations across both real-time and conventional applications. Furthermore, SMART also supports simultaneous prioritized and proportional resource allocation.

### 4.8.3. Hierarchical Scheduling

Because creating a single scheduler to service both real-time and conventional resource requirements has proven difficult, a number of hybrid schemes [7, 11, 23, 24, 68] have been proposed. These approaches attempt to avoid the problem by having statically separate scheduling policies for real-time and conventional applications, respectively. The policies are combined using either priorities [11, 23, 68] or proportional sharing [7, 24, 27] as the base level scheduling mechanism.

With priorities, all activities scheduled by the real-time scheduling policy are assigned higher priority than activities scheduled by the conventional scheduling policy. This causes all real-time activities, regardless of whether or not they are important, to be run ahead of any conventional activity. The lack of control results in experimentally demonstrated pathological behaviors in which runaway real-time computations prevent the user from even being able to regain control of the system [50].

With proportional sharing, a real-time scheduling policy and a conventional scheduling policy are each given a proportional share of the machine to manage by the underlying proportional share mechanism, which then time-slices between them. Real-time applications will not take over the machine, but they also cannot meet their time constraints effectively as a result of the underlying proportional share mechanism taking the resource away from the real-time scheduler at an inopportune and unexpected time in the name of fairness [25].

The problem with previous mechanisms that have been used for combining these scheduling policies is that they do not explicitly account for real-time requirements. These schedulers rely on different policies for different classes of computations, but they encounter the same difficulty as other approaches in being unable to propagate these decisions to the lowest-level of resource management where the actual scheduling of processor cycles takes place.

SMART behaves like a real-time scheduler when scheduling only real-time requests and behaves like a conventional scheduler when scheduling only conventional requests. However, it combines these two dimensions in a dynamically integrated way that fully accounts for real-time requirements. SMART ensures that more important activities obtain their resource requirements, whether they be real-time or conventional. In addition to allowing a wide range of behavior not possible with static schemes, SMART provides more efficient utilization of resources, is better able to adapt to varying workloads, and provides dynamic feedback to support adaptive real-time applications that is not found in previous approaches.

# 5 Implementation in a Commercial Operating System

To demonstrate the effectiveness of the SMART scheduler in a realistic general-purpose computing environment, we have implemented SMART in a commercial operating system, the Solaris operating system from Sun Microsystems. This is a UNIX SVR4 conformant operating system that is shipped with hundreds of thousands of computers each year. We used the most recent version of the operating system at the time of the prototype implementation, which was Version 2.5.1 of the operating system. The scheduling framework in this version of the operating system is largely the same as the improved Solaris scheduling framework that resulted from the work described in Chapter 2.

Our choice of using the Solaris operating system for our implementation was based on a few factors. First, it is a widely used general-purpose operating system, which gave us the opportunity to examine the impact of our scheduler on the performance of real applications in a commonly used computing environment. Second, the Solaris operating system is representative of other commercial operating systems in its structure and scheduling framework. Third, we had access to complete source code for the entire Solaris operating system. In addition, our previous experience with the operating system and the availability of kernel-level debugging tools were very helpful. Finally, the Solaris operating system was designed with some consideration for real-time requirements in mind. For instance, processes executing in kernel mode are usually executed at their respective user-level priorities and can be preempted, reducing the likelihood of priority inversion.

While there were a number of advantages to using a commercial operating system as the basis of our prototype implementation, there was also an implementation cost. Solaris is a full-featured operating system that is optimized for performance, not implementation ease.

Like many commercial operating systems, it is a large monolithic software system with hundreds of thousands of lines of code and little documentation. The interface between the scheduling framework and other aspects of the operating system was not always well-defined and parts of the system do not fully implement the interface for performance reasons. Furthermore, the system was not designed with a scheduler such as SMART in mind. As a result, the implementation of some aspects of SMART could not be done in the most straightforward and intuitive manner. Learning enough about this system to create the necessary interfaces between SMART and the Solaris operating system was a substantial cost in designing our prototype implementation.

This chapter discusses a number of the implementation issues in creating the SMART prototype in the Solaris operating system. Section 5.1 provides some necessary background about the Solaris scheduling framework. Section 5.2 describes our implementation methodology, which was to replace the existing Solaris dispatcher and introduce a new scheduling class. Section 5.3 describes the implementation of the SMART dispatcher, including improvements to the timer functionality of the operating system. Section 5.4 describes the implementation of the SMART scheduling class.

## 5.1. Solaris Scheduling Framework

The Solaris operating system is a multithreaded UNIX SVR4 conformant operating system. Unlike older UNIX systems which only provide kernel support for UNIX processes, Solaris scheduling is based on threads. These lightweight objects are less expensive to create and use than UNIX processes, and they can be independently scheduled. Threads are used in Solaris for executing applications, as well as for interrupt handling and other internal kernel functions. They are fully preemptible even when executing kernel code, allowing for better real-time responsiveness. In this section, we give an overview of the structure of the Solaris scheduler and describe some of the core thread scheduling mechanisms.

As previously mentioned in Chapter 2, the Solaris scheduler is a two-level UNIX SVR4 priority scheduling framework consisting of a set of scheduling classes and a dispatcher. Each thread is assigned to a single scheduling class. The job of each scheduling class is to make its own policy decisions regarding how to schedule threads assigned to the class. The job of the dispatcher is to merge the policy decisions of the different scheduling classes. It determines a global ordering in which to execute threads from all of the scheduling classes, and then performs the actual work of executing the threads according to that global ordering.

The scheduling classes and the dispatcher use priorities to perform their functions. When a thread is assigned to a scheduling class, a set of class scheduling parameters are associated with that thread. Associated with each scheduling class is a continuous range of class priorities. Using the class scheduling parameters of the thread and information about the execution history of the thread, the scheduling class determines a class priority for the thread. The class priority of a thread can change as the class scheduling parameters for the thread change, or as the execution history of the thread changes. Consider for instance the time-sharing (TS) class that comes as the default scheduling class for any UNIX SVR4 scheduler. A thread is assigned to the TS class with some nice value. The TS class determines a class priority for a thread by using the thread's nice value and information about how much processing time the thread has consumed recently. The TS class will periodically adjust the class priority of a thread depending on how much processing time the thread has consumed recently.

The policy decisions of the scheduling classes are merged by mapping their respective ranges of class priorities onto a continuous range of global priorities. The dispatcher then schedules threads based on these global priorities. Consider for instance two of the default scheduling classes that come with any UNIX SVR4 scheduler, the real-time (RT) class and the TS class. Since it is a UNIX SVR4 scheduler, the Solaris scheduler has global priorities 0-159. The class priorities of the RT class and the TS class each range from 0-59. However, the RT class priorities are mapped to global priorities 100-159 while the TS class priorities are mapped to global priorities 0-59. As a result, threads from the TS class are only executed if there are no threads from the RT class to execute.

The dispatcher uses a set of run queues to select threads for execution based on their global priorities. Each global priority value has a run queue associated with it. Since there are 160 global priorities in the Solaris scheduler, there are a corresponding set of 160 run queues.

When a thread is runnable, it is assigned to the run queue corresponding to its global priority. The dispatcher is called whenever the processor becomes available to execute a thread. To select a thread to execute, the dispatcher scans the run queues from highest to lowest priority and chooses the thread at the front of the first nonempty queue for execution. In other words, the highest priority runnable thread is selected for execution. Note that where the thread is placed on the run queue will impact when the thread is selected for execution by the dispatcher. The scheduling framework allows scheduling classes to determine where a thread should be placed on the run queue when it is runnable. A scheduling class can choose to place a thread at the back of a run queue or at the front of a run queue. For example, if a scheduling class always inserts threads at the back of run queues, then threads that were inserted earlier will run before threads that were inserted later. This will result in a First-In-First-Out scheduling policy.

In addition to determining the priority assignment of threads, the scheduling classes control how long a thread should be allowed to execute. A scheduling class may assign a time quantum to each thread. The time quantum defines the maximum amount of time that a thread can execute before the scheduler will preempt the thread and make another scheduling decision. Like other UNIX systems, this is enforced through the use of a periodic interrupt generated by a hardware clock. The interrupt calls a clock function for the respective scheduling class of the currently running thread. The function checks the execution time of the running thread and preempts the thread if it has used up its time quantum. If a scheduler class does not assign time quanta to its threads and does not support a clock function, threads belonging to the respective scheduling class will only be preempted by the dispatcher when a higher priority thread is available to run. Note that a thread will continue to run if it is higher priority than all other threads even if it has used up its time quantum.

While the scheduling framework provides several default scheduling classes, the framework is extensible. New scheduling classes can be implemented that are mapped to different global priority ranges. To support this extensible framework, an extensible system call is provided that allows users and applications to assign and change scheduling class parameters. These parameters can be assigned on a per thread basis. Each scheduling class takes the set of class parameters for a given thread and reduces it to a class priority and time quantum assignment for the thread.

## 5.2. Implementation Methodology

There are three basic implementation possibilities for the SMART prototype afforded by the Solaris scheduling framework. They are to implement SMART as a user-level process, to implement SMART just as a scheduling class in the existing Solaris scheduling framework, or to implement SMART as a replacement for the existing scheduling framework. We consider each of these possibilities in turn and describe the reasons behind our eventual choice of approach.

The easiest to implement would be to build the SMART prototype simply as a user-level process that runs at the highest priority defined by the scheduling framework. By running such a user-level scheduler process at the highest priority, the underlying scheduling framework simply hands control over to the user-level process. The process could then select a thread to execute based on the SMART scheduling algorithm, raise the priority of that thread to just below that of the scheduler process itself, then block itself for a period of time to allow the selected thread to execute. Not only would this approach be easier to implement because no kernel development is required, but it would be more portable as it could run on other priority-based schedulers, including other variants of UNIX and Windows NT. The primary problem with this approach is that scheduling events that are presented to the operating system would often not be known by the user-level process. For instance, the thread that is scheduled may run for less than the amount of time that the scheduler is blocked, in which case the scheduler would not be able to run when the scheduled thread completes its execution. The default Solaris dispatcher would then select a thread to run based on just priorities as opposed to the SMART algorithm.

Another implementation approach would be to build SMART as a new scheduling class in the Solaris scheduling framework. The framework provides a well defined interface for allowing developers to create their own scheduling policies which can be loaded on demand into the kernel. However, the framework does not allow the scheduling policies to directly select which thread to execute when the processor becomes available. Scheduling policies must indirectly impact this decision by the priorities they assign to their threads and the order in which they place those threads on the respective run queues. Furthermore, the framework only allows scheduling policies to insert threads at the front or the back of the run queues. These characteristics of the scheduling framework make it impossible to implement SMART as a scheduling class in an efficient manner. A key aspect of the SMART algorithm is its use of dynamic timing requirements in deciding which thread should be executed. This does not map well to the static priorities of the Solaris scheduling framework. In addition, SMART uses biased virtual finishing times for ordering threads of equal priority. This ordering function does not map well to the limited run queue ordering operations provided with the Solaris scheduling framework.

In light of the disadvantages of these other approaches, the approach that we choose for implementing our SMART prototype in the Solaris operating system was to replace the existing scheduling framework. Unlike a user-level implementation, this provided our prototype the same amount of access to information about scheduling events as the existing scheduling framework. Unlike a scheduling class implementation, this allowed us to define new run queue operations as necessary to support the scheduling mechanisms in SMART. In particular, we replaced the existing priority dispatcher with a SMART dispatcher that incorporates information about a thread’s deadlines and shares as well as priorities. Two challenges that we faced in this approach were providing interfaces between SMART and existing operating system code, and designing the SMART implementation in such a way to be backwards compatible with the existing scheduling framework. The latter goal is important for the purposes of providing an easy transition path for users of the existing scheduling framework to move to the SMART scheduling framework.

To provide a natural mapping to the structure of the Solaris operating system, threads are used as the basic schedulable entity in our SMART framework. For the SMART framework, we replaced the existing dispatcher and created a new scheduling class to provide access for users and applications to the new functionality of our system. These two aspects of the system are described in the next two sections.

## 5.3. Dispatcher Implementation

The SMART scheduler exploits a greater amount of information in making a scheduling decision than the existing Solaris scheduling framework. In particular, the SMART dispatcher makes use of more than just the single priority value associated with each thread in the Solaris scheduling framework. In addition to a priority, the dispatcher assumes that each thread is assigned a share, a bias, a deadline, and a time quantum. These parameters are determined by the scheduling classes and passed to the dispatcher. Default values are initially assigned for each parameter associated with a thread.

Our SMART dispatcher implementation maintains the same set of run queues as the Solaris scheduling framework, but uses them in a different way. Like Solaris, each run queue corresponds to a priority, and there are 160 priorities numbered 0-159. However, the run queues in SMART are not used directly for selecting which thread should execute. Instead, they are used for maintaining an importance ordering of all threads based on the respective thread priorities and biased virtual finishing times. Threads are assigned to the respective run queues based on priorities. Threads on the same run queue are ordered based on their biased virtual finishing times. When a thread needs to be selected for execution, the dispatcher starts from this importance ordered list of threads and uses the SMART algorithm to create a working schedule. The first runnable thread in the generated working schedule is then selected for execution.

Our SMART prototype uses the same function prototypes for inserting and removing threads from the run queues, but changes the underlying semantics of the functions. In the original framework, there were two queue insertion functions which respectively placed a thread at the front or back of a run queue. For our SMART framework, we need to be able to places threads in a run queue such that they are ordered by their respective biased virtual finishing times. To achieve this, we change the semantics of the queue insertion functions such that both functions now insert a thread in a run queue in biased virtual finishing time order. In the event that multiple threads have the same biased virtual finishing time, ties are broken based on the original semantics of the queue insertion functions. The queue insertion function that originally inserted at the front of the run queue will break ties by inserting a thread in front of any threads with the same biased virtual finishing time. The queue insertion function that originally inserted at the back of the run queue will break ties by inserting a thread in back of any threads with the same biased virtual finishing time. We will show later that this feature is used for backwards compatibility with the original Solaris scheduling framework.

The SMART dispatcher may run less important threads before more important threads when it determines that there is excess slack in the system. When a thread is selected for execution, the dispatcher should ensure that the thread only runs for an amount of time that still allows less urgent but more important threads to meet their timing requirements. As a result, the dispatcher must enforce a bound on the execution time given to a thread. This is done in our implementation by having the dispatcher set its own time quantum for a thread when the thread is selected to execute. The dispatcher itself will check to see that whether the time quantum of the thread has expired, in which case it will preempt the thread. This dispatcher time quantum is internal to the dispatcher and is separate from the time quantum used by the scheduling classes.

The standard UNIX SVR4 scheduling framework upon which the Solaris operating system is based employs a periodic 10 ms clock tick. It is at this granularity that scheduling events can occur, which can be quite limiting in supporting real-time computations that have time constraints of the same order of magnitude. In particular, the granularity of the time quantum parameter can be no smaller than the timer resolution. To allow a much finer resolution for scheduling events, we added a high resolution timeout mechanism to the kernel and reduced the time scale at which timer based interrupts can occur. The exact resolution allowed is hardware dependent, but was typically 1 ms for the hardware platforms we considered.

In our SMART implementation, the dispatcher is given the share and bias of each thread by the scheduling classes and uses that information to compute the biased virtual finishing time as each thread executes. The biased virtual finishing time is computed by the dispatcher because it serves as a global ordering function for threads from different scheduling classes that are at the same priority. This allows the SMART dispatcher to provide a proportional share abstraction to the scheduling classes, allowing for the creation of scheduling classes with different proportional share scheduling policies.

In addition to providing SMART functionality, the SMART dispatcher is designed to support legacy scheduling classes as well. This is done through a legacy scheduling class test function and proper definition of the default thread dispatcher parameters. All legacy scheduling classes are listed in an array provided to the legacy scheduling class test function. When a thread is assigned to a scheduling class, the test functions checks if the scheduling class is a legacy scheduling class. If so, it assigns the thread a set of default dispatcher parameters. These are the same defaults that are assigned when a thread is created. The priority is set to the same default as used with the original Solaris dispatcher, the share is set to zero, the bias is set to zero, the deadline is set to a maximum value, and the time quantum is set to a maximum value. If a thread has zero share, its biased virtual finishing time is set to a maximum value. As a result, all threads with nonzero shares will be enqueued in front of all threads with zero shares. More importantly, since all threads with zero shares will have the same maximum biased virtual finishing time, the tie breaking rules of the queue insertion functions will be used for those threads, reducing those functions to the original Solaris queue insertion functions. If a thread has a deadline set to the maximum value, the thread is considered a conventional thread. In particular, if all threads are conventional, the SMART dispatcher reduces to selecting the first runnable thread on the highest priority non-empty run queue. If a thread has a time quantum set to the maximum value, the time quantum is effectively ignored by the dispatcher. In summary, a thread with default values for its dispatcher parameters is scheduled in exactly the same way as the original Solaris dispatcher. This ensures that the SMART scheduling framework is backwards compatible with the original Solaris scheduling framework.

Legacy scheduling classes can be used at the same time as new scheduling classes written for the SMART scheduling framework. If all scheduling classes are mapped to nonoverlapping ranges of global priorities, the interaction of the scheduling classes in the SMART scheduling framework is similar to the standard UNIX SVR4 framework. If a new scheduling class and a legacy scheduling class are mapped to overlapping ranges of global priorities, the SMART framework gives preference to the new scheduling class over the legacy scheduling class for threads at the same priority. This is because threads in legacy scheduling classes were each assigned zero shares with a corresponding maximum biased virtual finishing time. This causes such threads to be considered after threads in new scheduling classes which are each assigned non-zero shares and a smaller biased virtual finishing time. The choice of favoring new scheduling classes over legacy scheduling classes at the same priority level was to some degree an arbitrary one; the reverse could also have been done. Both choices would provide support for new SMART functionality and legacy scheduling class functionality.

## 5.4. Class Implementation

In addition to supporting legacy scheduling classes, the SMART dispatcher provides new functionality that can be exploited through the creation of new scheduling classes. For our SMART prototype, we also created a SMART scheduling class. The primary purpose of this class is to support the SMART scheduling interface for users and applications. The class not only ensures that scheduling parameters provided from users and applications are valid, but it also sets default parameters when such information is not provided. The user and application interfaces are based on the Solaris priocntl system call, an extensible interface for setting and reading scheduling class parameters.

In our implementation, the SMART scheduling class is also responsible for automatically adjusting the bias associated with conventional threads. This is done in a table-driven manner using a UNIX SVR4 scheduler mechanism that is supported in the Solaris operating system. This mechanism was originally designed to support the multi-level feedback discipline used by the TS scheduling class. The TS class reads in a scheduling table with entries corresponding to priority levels. Each entry specifies a priority, a time quantum that is to be assigned to a thread at the given priority, and the priority to assign a thread at the given priority when its time quantum is used up. As a thread executes and completes its time quanta, it is reassigned a new priority after each time quantum completion, with the assigned priorities monotonically decreasing. Instead of using this scheduling table to adjust priorities, the SMART scheduling class uses a scheduling table to adjust biases. Each entry in the SMART scheduling table specifies a bias, a time quantum that is to be assigned to a thread at the given bias, and the bias to assign a thread at the given priority when its time quantum is used up. As a thread executes and completes its time quanta, it is reassigned an increasing bias after each time quantum completion.

For real-time threads, the SMART scheduling class implements the notification mechanism that is used for informing real-time applications when their deadlines cannot be met. This is done using the basic timeout and signal mechanism in standard UNIX SVR4 systems. When a thread specifies a notify-time with its time constraint, the scheduler sets a timeout corresponding to the notify time. The timeout causes a clock interrupt to go off at the prescribed time. A flag is set once the notify time expires. Once the flag is set, then each time the thread is selected for execution, the scheduling class checks to see if the thread will meet its deadline. This requires that the scheduling class be informed when a thread belonging to the class is selected for execution by the dispatcher. The original UNIX SVR4 scheduling framework had no such way of doing this, so a new class function was added in the SMART scheduling framework. This class function is called when a thread from the respective scheduling class is selected for execution. The function can be used for performing necessary operations on the thread that has been selected for execution, including checking to see if the deadline will be met. If the thread will not be able to meet its deadline, a signal is sent to the process indicating that the deadline will not be satisfied. A previously unused signal number is used to distinguish the notification signal from other signals that the process may receive. If the process does not have a signal handler installed, the notification signal is ignored.

## 5.5. Summary

Although the Solaris scheduling framework was not designed to support the demands of multimedia applications, we have been able to extend the framework to implement the SMART scheduler in the Solaris operating system. This prototype implementation demonstrates that it is possible to include SMART support for multimedia applications in the context of currently available general-purpose operating systems. It also demonstrates that SMART functionality can be implemented in a way that continues to provide backwards compatibility for legacy scheduling policies.

We note that while the SMART Solaris prototype is not commercially available, its implementation has been quite robust and supports all the same applications and system functions as the standard Solaris operating system. We have run the SMART Solaris prototype on a day to day basis, and have used it to run many of the applications that were needed for writing this dissertation itself.

# 6 Measurement and Performance

To evaluate the effectiveness of the SMART scheduler, we conducted a number of experiments on our SMART prototype, running microbenchmarks as well as real, commercial applications. Because of the complex interactions between applications and operating systems in general-purpose computer systems, we placed an emphasis on evaluating SMART with real applications in a fully functional system environment. Our experiments included quantitative measurements of application and system performance for multimedia, interactive, and batch applications running on SMART in head-to-head comparisons with other schedulers used in commercial and research settings.

This chapter describes our experimental results and is organized as follows. Section 6.1 describes the experimental testbed which was used for our experiments. Section 6.2 describes experiments with various microbenchmarks that demonstrate the range of behavior possible with the scheduler in a real system environment. Section 6.3 examines the performance of a set of commercially available multimedia video applications running on SMART versus UNIX SVR4, focusing particularly on the benefits that result from the SMART interface. Section 6.4 compares the performance of SMART against other schedulers commonly used in practice and research by running real applications and measuring the resulting behavior. This comparison considers not only real-time multimedia application performance, but also quantitatively measures the performance of interactive and batch applications.

## 6.1. Experimental Testbed

The experiments were performed on a standard, production SPARCstation™ 10 workstation with a single 150 MHz hyperSPARC™ processor, 64 MB of primary memory, and 3

GB of local disk space. The testbed system included a standard 8-bit pseudo-color frame buffer controller (i.e., GX). The display was managed using the X Window System. The Solaris 2.5.1 operating system was used as the basis for our experimental work. In particular, the high resolution timing functionality described in Section 5.3 was used for all of the schedulers in our experiments to ensure a fair comparison. On the testbed workstation used for these experiments, the timer resolution was 1 ms.

All measurements were performed using a minimally obtrusive tracing facility that logs events at significant points in application, window system, and operating system code. This is done via a light-weight mechanism that writes timestamped event identifiers into a memory log. The timestamps are at 1 μs resolution. We measured the cost of the mechanism on the testbed workstation to be 2-4 μs per event. We created a suite of tools to post-process these event logs and obtain accurate representations of what happens in the actual system.

All measurements were performed on a fully functional system to represent a realistic workstation environment. By a fully functional system, we mean that all experiments were performed with all system functions running, the window system running, and the system connected to the network. At the same time, an effort was made to eliminate variations in the test environment to make the experiments repeatable. To this end, the testbed system was restarted prior to each experimental run.

## 6.2. Microbenchmarks

We highlight some performance results for various mixes of real-time and conventional resource requests. These requests were generated using a set of simple applications that allow us to vary their resource requirements to demonstrate the effectiveness of SMART under a variety of workloads. We look at conventional activities, real-time activities, and a combination of both in Sections 6.2.1, 6.2.2, and 6.2.3 respectively. In particular, we focus on the proportional sharing aspects of SMART. We demonstrate that proportional sharing is achieved for all the cases, regardless of whether the real-time requests present (if any) have overloaded the system. We show in both Sections 6.2.2 and 6.2.3 that the scheduler drops the minimum number of deadline requests to achieve fair sharing, in the case of overload. Finally, we also show in Section 6.2.3 that latency tolerance helps minimize the number of deadlines dropped.

### 6.2.1. Conventional Applications

The first case presented is based on the execution of three identical conventional compute-oriented applications, *C1*, *C2*, and *C3*, with relative shares of the ratio 3 : 2 : 1. The applications were started at approximately the same time and have a running time of about 338 seconds. We logged the cumulative execution time of each application versus wall clock time. The results are shown by the solid line curves in Figure 6-1. We note that Figure 6-1 as well as all of the other figures simply present the raw sampled data with no interpolation between sample points.

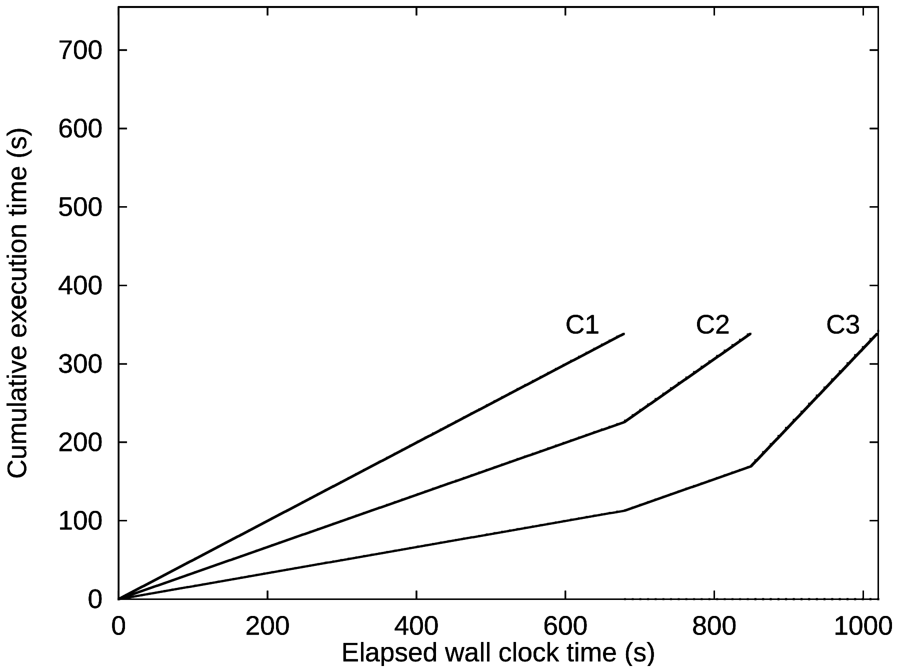


Figure 6-1. Execution times for conventional applications when proportional sharing with SMART shares 3:2:1

If the system were perfect, we would expect the following finishing times:

Activity *C1*: $338 \times 6/3 \ = \ 676$ seconds

Activity *C2*: $676 + (338 - (676 \times 2/6)) \times 3/2 \ = \ 846$ seconds

$$\text{Activity } C3\text{: } 676 + 170 + (338 - (676 \times 1/6) - 170 \times 1/3) = 1014 \text{ seconds.}$$

The corresponding ideal performance curves are also plotted as dotted lines in Figure 6-1. However, the dotted ideal curves are hardly visible because of the close match with the actual experimental results. The results match the expected behavior very well, with activities *C1*, *C2*, and *C3* finishing at times 678, 848, and 1018 seconds, respectively. Furthermore, the slopes of the graphs, indicating the resource consumption rates, show that the activities are serviced in a proportionally fair manner throughout the execution. As expected, the slopes of *C1*, *C2*, and *C3* are of the ratio 3.00 : 2.00 : 1.00 in the first stage of the computation when all the programs are running. The slopes of *C2* and *C3* are of the ratio 2.00 : 1.00 in the second stage when activities *C2* and *C3* are running.

### 6.2.2. Real-time Applications

#### 6.2.2.1. Optimal Performance in Underload

To demonstrate the performance of SMART for real-time applications in underload, we executed two real-time applications, *R1* and *R2*, with periodic resource requests. The resource requests were event-driven, with the event arrival interval determining the deadline of the respective request. All of the real-time applications used in these experiments employed the time constraint interface in SMART and were notified immediately if their deadlines could not be met. If a deadline could not be met, the application discarded the corresponding resource request and waited until the arrival of the next event to be processed. We note that for the particular real-time applications used in these experiments, there was a small amount of execution time variation for each resource request when multiple real-time applications were executed together. This was caused by cache conflicts between the applications, which depended on the exact order in which the resource requests were processed.

*R1* required 28-30 ms of execution time to complete each resource request, with each resource request having a 40 ms deadline from its instantiation. *R2* required 18-20 ms of execution time to complete each resource request, with each resource request having a 90 ms deadline from its instantiation. *R1* was given 2000 events to process and *R2* was given

888 events to process. Given these resource requirements, the system can be kept busy about 97% of the time running *R1* and *R2* during a period of about 80000 ms. *R1* and *R2* were both assigned equal shares, though the assignment of shares makes no difference in this case. We logged the number of missed deadlines and the cumulative execution time obtained by each activity, which is illustrated by the solid line curves in Figure 6-2.

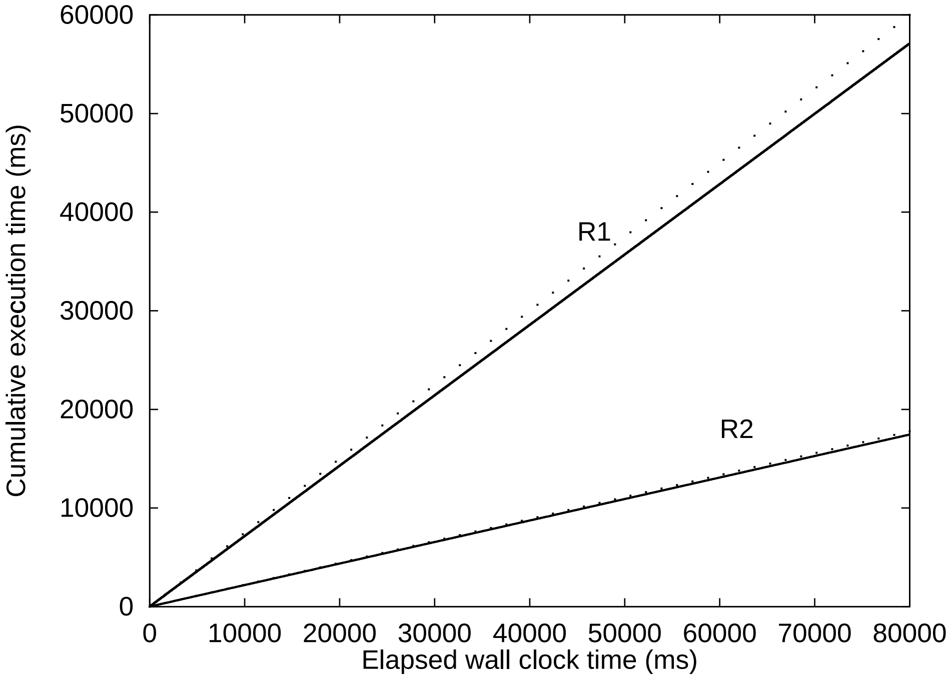


Figure 6-2. Execution times for real-time applications when using SMART in system underload

If the system were perfect, we would expect all deadlines to be satisfied; both activities should be able to obtain their requested resources. Therefore, each activity should be able to obtain resources proportional to their respective resource requests. If *R1* required exactly 30 ms of execution time for each 40 ms deadline and *R2* required exactly 20 ms of execution time for each 90 ms deadline, we would expect both activities to finish in roughly 800 seconds with the following cumulative execution times and percentages of processing time:

Activity *R1*: $2000 \times 30 = 60000$ ms, $60000 / 80000 = 75\%$,

Activity *R2*: $888 \times 20 = 17760$ ms, $17760 / 80000 = 22\%$.

The corresponding ideal performance curves are represented by the respective dotted lines in Figure 6-2. We can see in Figure 6-2 that both applications obtained nearly their ideal resource consumption rates, with the slight deviation from ideal being due to each activity on average requiring slightly less than their respective ideal execution times to complete their resource requests. Activity *R1* obtained 57,066 ms of execution time and activity *R2* obtained 17,426 ms of execution time. This translates into *R1* receiving 71% of the processing time and *R2* receiving 22% of the processing time. The slopes of the plots Figure 6-2 shows that each activity receives their respective processing time in a consistent manner throughout their execution. More importantly, because SMART is optimal in underload, there were no missed deadlines for either application.

To show that the optimal performance of SMART in underload translates well into practice, we also demonstrate that SMART performs well in meeting the deadlines of real-time applications when the system utilization is nearly 100%. We executed two periodic real-time applications, *R1* and *R3*, but this time the applications together consumed nearly 100% of the machine. *R1* again required 28-30 ms of execution time to complete each resource request, with each resource request having a 40 ms deadline from its instantiation. *R3* required 18-20 ms of execution time every 80 ms. *R1* was given 2000 events to process and *R3* was given 1000 events to process. *R1* and *R3* were both assigned equal shares, though the assignment of shares makes no difference in this case. We logged the number of missed deadlines and the cumulative execution time obtained by each activity, which is illustrated by the solid line curves in Figure 6-3.

If the system were perfect and the actual processing time required in any given interval never exceeds 100% utilization, we would expect all deadlines to be satisfied in this case as well; both activities should be able to obtain their requested resources. Furthermore, if *R1* required exactly 30 ms of execution time for each 40 ms deadline and *R3* required exactly 20 ms of execution time for each 80 ms deadline, we would expect both activities to finish in roughly 800 seconds with the following cumulative execution times:

Activity *R1*: $2000 \times 30 = 60000$ ms

Activity *R3*: $1000 \times 20 = 20000$ ms.

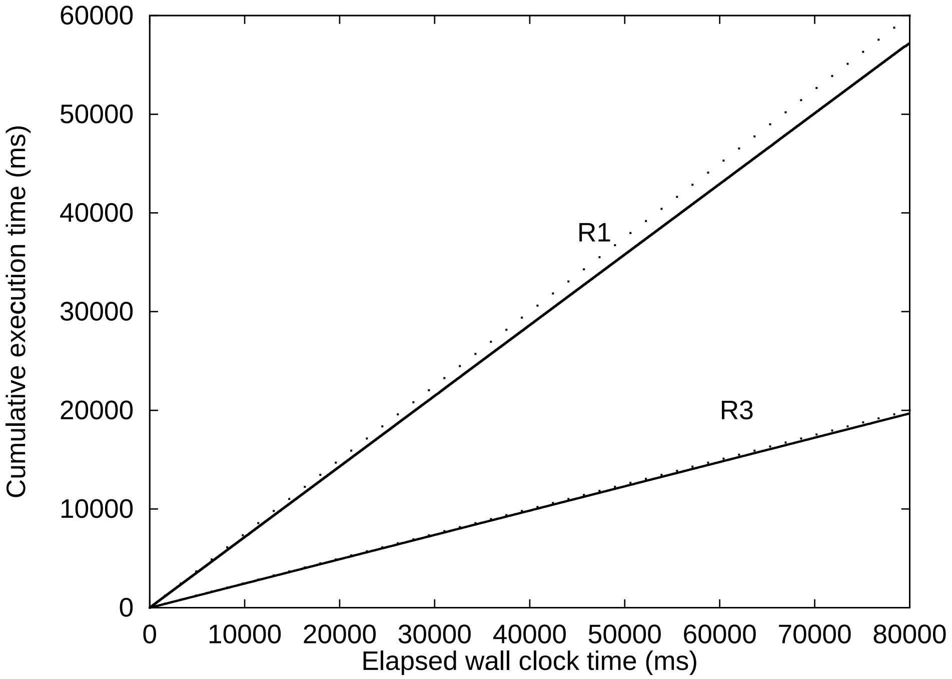


Figure 6-3. Execution times for real-time applications when using SMART on a nearly 100% loaded system

The corresponding ideal performance curves are represented by the respective dotted lines in Figure 6-3. We can see in Figure 6-3 that both applications obtained nearly their ideal resource consumption rates, with the slight deviation from ideal being due to each activity on average requiring slightly less than their respective ideal execution times to complete their resource requests. Activity *R1* obtained 57,178 ms of execution time and activity *R3* obtained 19,673 ms of execution time. SMART met all but 3 of the 1999 deadlines of *R1* and met all of the deadlines of *R3*. Even with nearly 100% average resource utilization and some variability in the execution times of these applications, SMART was able to meet over 99% of the deadlines.

### 6.2.2.2. Proportional Sharing in Overload

To demonstrate the unique ability of SMART to allow real-time applications to proportionally share resources in overload, we executed three identical real-time applications, *R1, R2*, and *R3,* with relative shares of the ratio 3 : 2 : 1. Each application takes 18-20 ms of execution time to complete each resource request, and each resource request has a 40 ms deadline from its instantiation. To show the dynamic behavior of these applications when the application mix changes, the applications are started at approximately the same time, but each application is executed for a different number of iterations. *R1* processed a sequence of 1000 real-time requests, *R2* processed a sequence of 1500 real-time requests, and *R3* processed a sequence of 2000 real-time requests. We logged the cumulative execution time and number of deadlines missed for each application. These measurements are illustrated by the solid line curves in Figures 6-4 and 6-5.

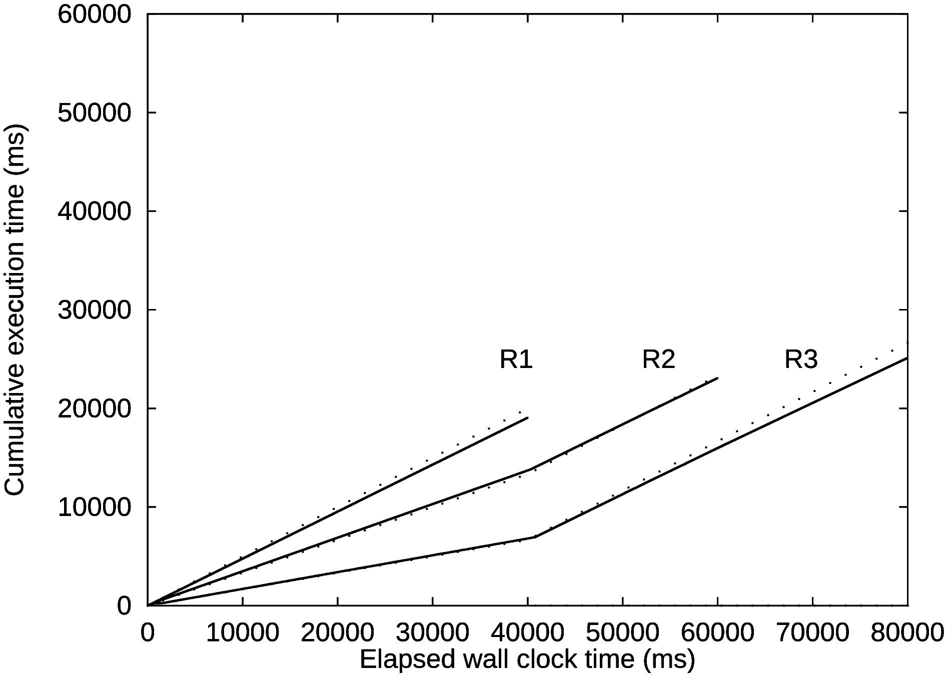


Figure 6-4. Execution times for real-time applications when proportional sharing with SMART shares 3:2:1 in system overload and underload

If the system were perfect, we would expect the three activities to accumulate processing time in accordance with their shares during the first 40,000 ms, then at the completion of *R1*, *R2*, and *R3* should be able to complete the remainder of their deadlines since the system is no longer overloaded. In particular, the ideal total cumulative execution times would be:

Activity *R1*: $40000 \times 3/6 = 20000$ ms

Activity *R2*: $40000 \times 2/6 + (1500 - 1000) \times 20 = 23333$ ms

Activity *R3*: $40000 \times 1/6 + (2000 - 1000) \times 20 = 26667$ ms.

The corresponding ideal performance curves showing cumulative execution time for each activity are plotted as dotted lines in Figure 6-4. As the system is overloaded during the first 40,000 ms, not all deadlines can be met. Based on their respective resource requirements and shares, we would expect the total number of deadlines met by each activity to be:

Activity *R1*: $20000 / 20 = 1000$ out of 1000 deadlines met

Activity *R2*: $23333 / 20 = 1166$ out of 1500 deadlines met

Activity *R3*: $26667 / 20 = 1333$ out of 2000 deadlines met.

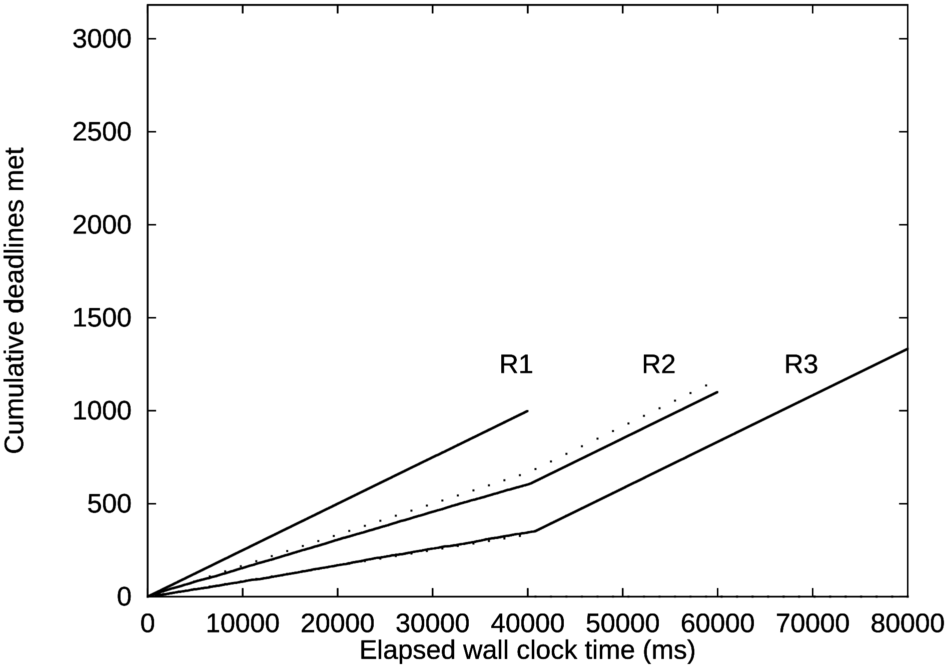


Figure 6-5. Deadlines met by real-time applications when proportional sharing with SMART shares 3:2:1 in system overload and underload

The performance curves showing the ideal number of deadlines met for each activity are plotted as dotted lines in Figure 6-5. These ideal numbers assume not only that each activity receives the ideal amount of resources, but also that all of that processing time can be effectively used toward meeting deadlines; processing time is not wasted on deadlines that will not be met. To do this, it is important to not only allocate processing time in the right sized quanta for each activity, but also that the quanta be provided at the right time to be synchronized with the deadlines.

We can see there is a close match between the measured results and the ideal curves. The total cumulative execution times of *R1*, *R2*, and *R3* were 19,045, 23,061, and 25,094 ms, respectively. The total number of deadlines met by *R1*, *R2*, and *R3* were 999 out of 1000, 1100 out of 1500, and 1331 out of 2000, respectively. The difference between the total number of expected deadlines met and the actual number of deadlines met is less than 2%.

We divide the results into the overloaded and underloaded periods. During the first 40 seconds of elapsed wall clock time, all three applications are executing. The slopes of the left graph show the respective measured resource consumptions of *R1*, *R2*, and *R3* to be in the ratio 2.80 : 2.02 : 1.00. Note that the resource consumption of *R1* is a bit less than its proportional share because it does not require its proportional share to complete its deadlines. If the system were perfect and that there are no variability in the activities' execution time, we expect applications *R1*, *R2*, and *R3* to meet roughly 100%, 66%, and 33% of their respective deadlines. During this period, each application had 1000 deadlines that it desired to have satisfied. Figure 6-5 shows the number of deadlines met. During the first 40 second period, *R1* met 999 deadlines, *R2* met 604 deadlines, and *R3* met 345 deadlines. They correlate well with the ideal values.

*R1* completes its execution at the beginning of the next 20 seconds of wall clock time while *R2* and *R3* continue to compete for resources. Note that the remaining activities are no longer requesting more than their proportional shares. The slopes of Figure 6-4 show the respective measured resource consumptions of *R2* and *R3* during this period to be in the ratio 1.02 : 1.00. Both activities miss a couple of deadlines as R1 is completing its execution, but then are able to satisfy all of their remaining deadlines. This example shows SMART automatically adjusts to the load condition and transitions between overload and underload gracefully.

### 6.2.3. Conventional and Real-time Applications

#### 6.2.3.1. Real-time Requests Using Less than Proportional Share

In mixing conventional and real-time applications, we first consider the case when the real-time applications require less than their proportional share of resources to satisfy their deadlines. We show two examples of this case in Figures 6-6 and 6-7 using applications with different resource requirements and different share assignments.

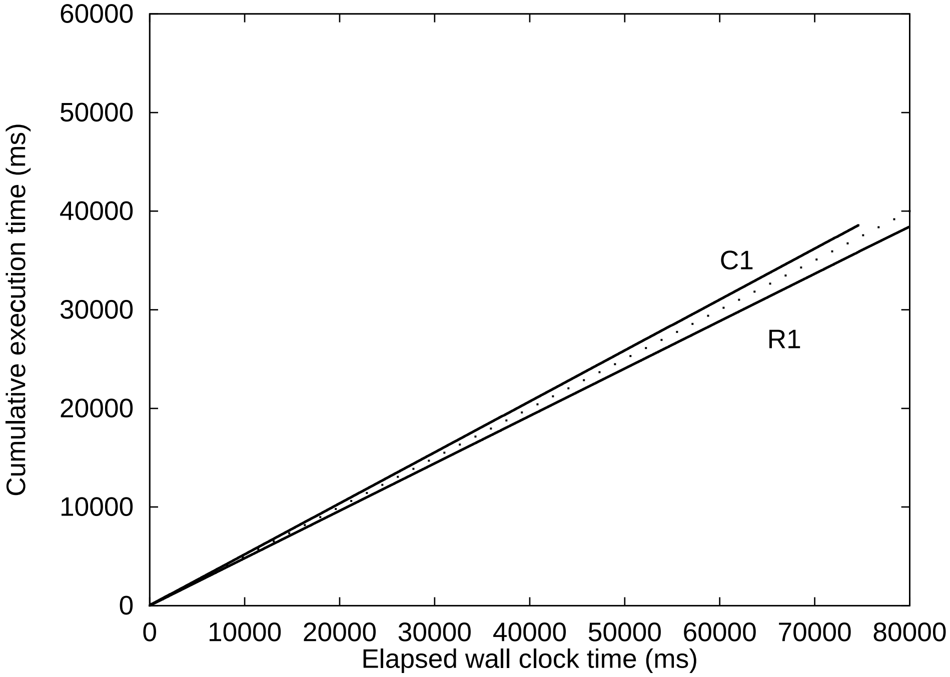


Figure 6-6. Execution times for real-time and conventional applications when proportional sharing with equal SMART shares

Figure 6-6 shows two equal-share applications, *R1* and *C1*, of which *R1* is real-time and *C1* is conventional. In particular, *R1* required 18-20 ms of execution time for each of its resource requests, and has a 40 ms deadline from its instantiation. If the system were perfect and *R1* required exactly 20 ms to process each 40 ms deadline request, we would expect *R1* not to miss any deadlines and for each application to use 50% of the system, as illustrated by the dotted line in Figure 6-6. Our measured results correspond well with the ideal and indicate that *R1* did not miss any of its deadlines. Since *R1* actually required slightly less than its proportional share of resources, *C1* was allowed to consume more than its proportional share and make better progress. In total, *R1* consumed roughly 48% of the system while *C1* consumed the remaining 52% of the system.

Figure 6-7 shows a periodic real-time application *R2* and a conventional application *C2* whose respective shares are in the ratio 3 : 1. *R2* required 28-30 ms of processing time with a periodic deadline every 40 ms. If the system were perfect and *R2* required exactly 30 ms to process each 40 ms deadline request, we would expect *R2* not to miss any deadlines and for *R2* and *C2* to consume resources in proportion to their shares, 75% and 25%, respectively, as illustrated by the dotted lines in Figure 6-7. Our measured results correspond well with the ideal and indicate that *R2* did not miss any of its deadlines. *R2* consumed roughly 72% of system while *C2* consumed the remaining 28% of the system. The slight difference from ideal is due to the fact that *R2* consumes a little bit less than the stated ideal.

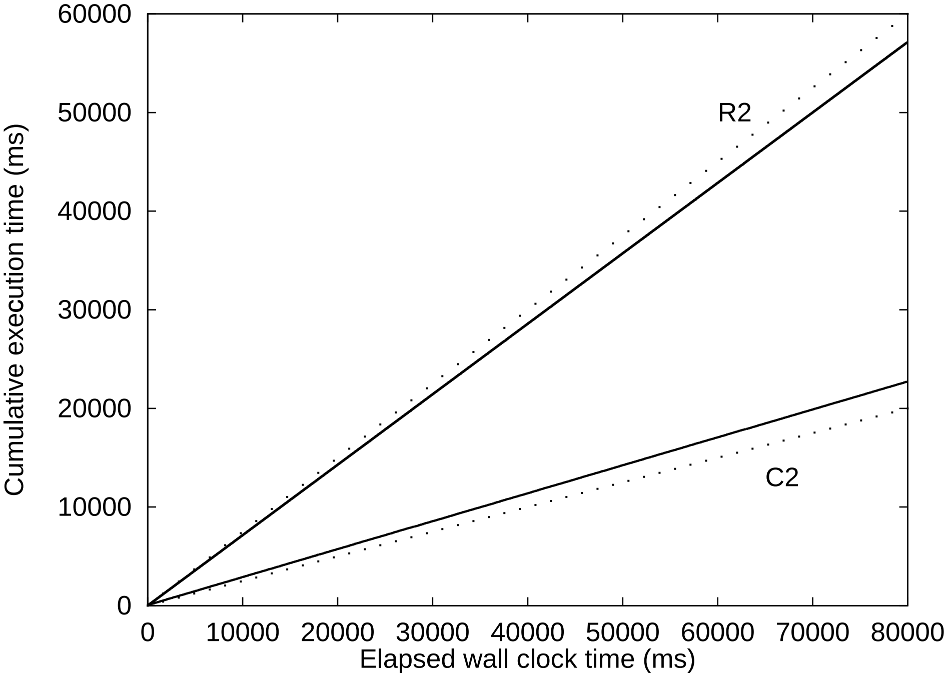


Figure 6-7. Execution times for real-time and conventional applications when proportional sharing with SMART shares 3:1

#### 6.2.3.2. Latency Tolerance

To show some of the benefit of latency tolerance, we again consider the case of two equal-share applications, *R1* and *C1*, of which *R1* is real-time and *C1* is conventional. In this case however, while each resource request of *R1* still has a 40 ms deadline from its instantiation, the work required for each resource request varies with an even distribution between 10-30 ms of execution time, with an average of 20 ms. While the resource consumption graph in this case remains similar to Figure 6-6 given the time scales involved, the number of deadlines that are missed by *R1* depends on the latency tolerance of *C1*. If *C1* cannot tolerate any latency, then *R1* misses 192 out of 1999 deadlines. However, if *C1* is given a latency tolerance of just 100 ms, *R1* can execute without missing any of its deadlines, despite the fact that its desired resource consumption for any given resource request varies from 25% to 75% of the machine.

#### 6.2.3.3. Proportional Sharing with Latency Tolerance in Overload

We demonstrate that SMART is able to share resources proportionally even for mixes of conventional and real-time applications in which the set of real-time resource requests is overloaded. Figures 6-8 and 6-9 show the results of executing three equal-share applications, *R1*, *R2*, and *C1*, of which *R1* and *R2* are real-time and *C1* is conventional. In particular, *R1* and *R2* are identical applications, each requiring roughly 38-41 ms of execution time for each resource request, and each resource request having a 60 ms deadline from its instantiation. Each real-time activity processed a sequence of 2000 resource requests. *C1* is a conventional activity with a latency tolerance of 200 ms and takes 40,000 ms of processing time to complete.

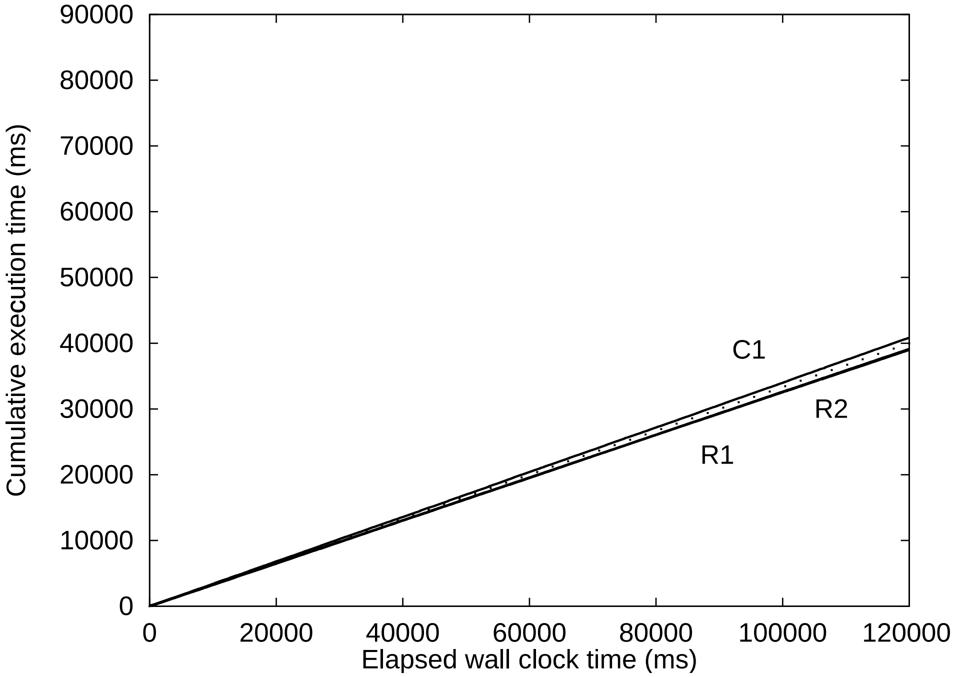


Figure 6-8. Execution times for real-time and conventional applications when proportional sharing with equal SMART shares in system overload

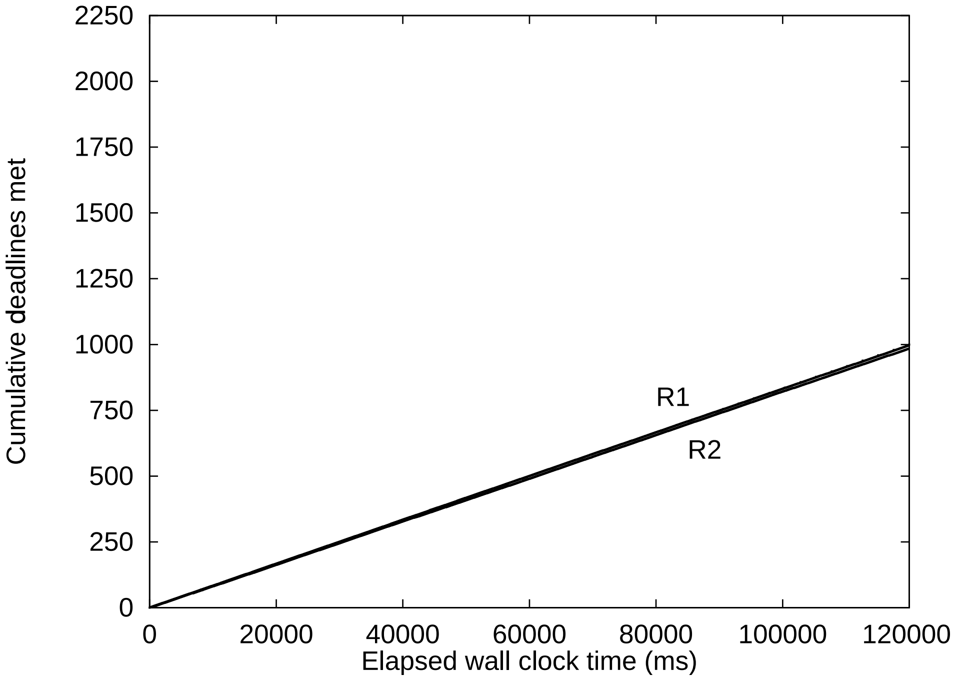


Figure 6-9. Deadlines met by real-time applications when proportional sharing with conventional applications with equal SMART shares in system overload

If the system were perfect, we expect activities to share resources based on their shares since the system is overloaded. As a result, each activity should be allocated 1/3 of the resources. All three activities should complete in 120,000 ms. Their ideal performance curves fall on the dotted line in Figure 6-8. Given 1/3 of the resources, each real-time activity should be able to complete half of its deadlines:

Activities *R1*, *R2*: $2000 \times (60/40) \times 1/3 = 1000$ out of 2000 deadlines met.

Figure 6-8 shows that all three applications consume nearly identical proportions of the resource, in close correspondence with the ideal. *C1* was able to consume a slightly larger portion of the processing time because it was always able to run if there were available resources. *R1* and *R2* were only able to make valuable use of the processing time when it was possible to satisfy their respective deadlines. For instance, if *R1* still had to run 40 ms to meet its deadline and the deadline was only 30 ms away, it would be useless to run *R1*.

Figure 6-9 shows that the number of deadlines met by *R1* and *R2* correspond well with their expected values. Out of 2000 deadlines, *R1* met 985 deadlines while *R2* met 998 deadlines.

The number of respective deadlines that each of them completed is in the ratio 1.00 : 1.01, which very closely corresponds to their shares. Over the course of 120 seconds of execution, the total amount of time spent by both *R1* and *R2* in processing deadlines that could not be met was less than 1 second. The SMART scheduler is able to make efficient use of processor cycles to devote the processor allocations of both real-time applications almost entirely toward resource requests whose deadlines can be met.

The ability of the conventional application *C1* to tolerate latency was a contributing factor in the ability of both real-time applications to meet such a large percentage of their deadlines, given their proportional shares, variances in execution times, and the overloaded condition. At the same time, the conventional application was still able to consume slightly more than its proportional share of processor cycles due to its ability to run whenever there were cycles that could not be used effectively by the real-time applications. When the latency tolerance of *C1* was instead zero, the number of deadlines missed increased by 10%.

### 6.2.4. Summary of Microbenchmark Results

Our measurements of microbenchmark real-time and conventional applications running on SMART demonstrate that SMART can provide proportional sharing across both real-time and conventional applications, even as the load on the system changes from underload to overload. Our empirical measurements correlate well with expected behavior, with each application obtaining nearly its ideal resource allocation in each experiment. Furthermore, our results also demonstrate that SMART is able to make effective use of the proportional share allocation given to a real-time application in meeting the deadlines of the application, even when the system is overloaded.

## 6.3. Commercial Multimedia Video Applications

While microbenchmark performance is commonly used as a basis for evaluating a scheduler, the real test of the effectiveness of a scheduler is its performance on mixes of real applications. Real applications are much more complex than microbenchmarks. This is especially the case with multimedia video applications, which often require intensive processor as well as memory operations. They also often need access to specialized input and output devices in conjunction with their need for processor cycles.

To provide an evaluation of SMART with real applications, we conducted experiments on a mix of commercial multimedia video applications. We present actual measurements from these experiments to demonstrate how the performance of multimedia applications can be improved using the SMART interface. Because UNIX SVR4 serves as a common basis for workstation operating systems, we have compared the performance of multimedia applications when using SMART against the performance of these applications when using the standard UNIX SVR4 schedulers. In particular, our experiments evaluate three different schedulers: SMART, UNIX SVR4 time-sharing (TS) scheduling, and UNIX SVR4 real-time (RT) scheduling.

As a representative multimedia video application, we used the SLIC-Video player, a low-cost video product that captures and displays video images in real-time. As a baseline, we measure the performance of the application when running on an otherwise quiescent system. We then measure the performance of multiple SLIC-Video players running under UNIX TS and UNIX RT with a dynamically changing load. To improve the video performance, we describe a few simple modifications to the application that allow it to take advantage of the SMART interface, then present results to quantify the performance improvement achieved.

### 6.3.1. Application Description and Quality Metric

SLIC-Video is a hardware/software video product used in Sun Microsystems workstations. The SLIC-Video hardware consists of an SBus I/O adaptor that permits the decoding and digitization of analog video streams into a sequence of video frames. This video digitizing unit appears as a memory-mapped device in an application’s address space and allows a user-level application to acquire video frames. The SLIC-Video player software consists of an application that captures the video data from the digitizer board, dithers to 8-bit pseudo-color in the case of a system with a standard 8-bit pseudo-color frame buffer controller, and directly renders the pixels to the frame buffer while coordinating with the X window server for window management. The resolution of the image rendered is configurable by the application. For our experiments, the image rendered was selected to be a standard size of 320 x 240 pixels.

The digitizer board has a limited amount of buffering that allows the hardware to continue to process an analog video stream into video frames while the software captures video data from the digitizer board. The buffer has three slots that are organized as a ring; when it is full, the hardware wraps around to the beginning of the buffer and overwrites its contents. Each slot is assigned a timestamp when it is written. Locking is used to ensure that the hardware does not overwrite a buffer slot that is being read by the software and the software does not read a buffer slot that is in the middle of being written by the hardware. In normal playback mode, the hardware digitizer cooperates with the software capture by sending a signal each time the hardware completes digitizing a frame. The signal is synchronized with the arrival rate of frames and occurs one full frame time after the capture board first begins digitizing a frame. Upon receiving the signal, the software follows a policy of reading from the buffer slot with the earliest timestamp. For our experiments, the arrival rate of frames from the hardware to the software is 29.97 frames per second (fps). As a result, the software begins processing each digitized frame roughly 33 ms after the hardware first begins digitizing the corresponding frame of analog video.

To describe the performance of the video application, we first discuss a metric for measuring the quality of its results. In video playback, the first goal in delivering the highest quality video is to preserve the temporal alignment of the incoming video stream. The time delay between frame arrival and frame display should be fairly constant. In addition to constant time delay, it is desirable to have constant interdisplay times between displayed frames. We would like to have all of the incoming frames rendered if possible. If many of the incoming frames cannot be rendered on time, it is desirable to discard frames in a regularly spaced fashion for more constant interdisplay times as opposed to discarding them unevenly. This provides better video quality especially for high-motion scenes. In particular, uncertainty is worse than latency; users would rather have a 10 fps constant frame rate as opposed to a frame rate that varied noticeably from 2 fps to 30 fps with a mean of 15 fps. Finally, for a mix of video players, it is desirable to allow the user to bias the performance of the applications in accordance with his preferences.

For our experiments, three SLIC-Video capture cards were added to the test system to permit the execution of three video players showing three different video sources at the same time. The video sources used were a video camera and television programming from a Sun Tuner. Video was displayed to the standard 8-bit pseudo-color frame buffer used on the testbed system. As described earlier, the display was managed using the X Window System.

### 6.3.2. Baseline Performance

In preparation for our discussion on the performance of multiple SLIC-Video players running on different schedulers, we first measured the performance of the SLIC-Video player running by itself on an otherwise quiescent system. The application was executed and measured for a 300 second time period. The application characteristics measured were the percentage of CPU used, the percentage of frames displayed, the average and standard deviation in the delay between the arrival and display of each frame, and the average and standard deviation in the time delta between frames being displayed. The standard deviation in the delay between frame arrival and frame display is the primary measure of quality. It is indicative of how well the temporal alignment in the video stream is preserved. The standard deviation in the time delta between frame displays is a secondary measure of quality. It measures the variability in the interdisplay times. Separate measurements were made for each 100 second execution interval of the application. These measurements are shown in Table 6-1. We performed these measurements using the UNIX SVR4 TS scheduler, the UNIX SVR4 RT scheduler, and the SMART scheduler. There was no significant difference in the baseline measurements for different schedulers running the single video application.

| *elapsed time* | *CPU usage* | *frames played* | *avg delay* | *std delay* | *avg delta* | *std delta* |
|---:|---:|---:|---:|---:|---:|---:|
| 0-100s | 87.28% | 99.73% | 64.15 ms | 1.84 ms | 33.43 ms | 2.48 ms |
| 100-200s | 87.32% | 99.80% | 65.17 ms | 1.93 ms | 33.41 ms | 1.80 ms |
| 200-300s | 87.35% | 99.83% | 66.19 ms | 1.27 ms | 33.40 ms | 1.98 ms |

Table 6-1. Baseline application performance

The measurements show that SLIC-Video uses up nearly 90% of the CPU to display video at 29.97 320x240 pixel fps. It displays over 99% of the frames that arrive, and it does so in a timely manner that preserves the temporal alignment in the video stream. Both the delay between frame arrival and frame display, and the time delta between frames displayed have minimal variation. The delay between frame arrival and frame display is roughly two frame times, corresponding to the application policy of processing the frame in the digitizer hardware buffer with the earliest timestamp. This delay is due to the software not receiving the frame from the hardware digitizer until at least one frame time after the frame arrives to the hardware, and the software requiring roughly one frame time to process each video frame. The time delta between displayed frames is one frame time, corresponding to the fact that over 99% of the frames that arrive are displayed.

### 6.3.3. Scheduler Experiments and Measurements

We examine the impact of different schedulers on the performance of multiple SLIC-Video players running under a dynamically changing load. The scenario we used was to first run two video players V1 and V2 for 100 seconds, then start the execution of a third video player V3 and run all three video players for the next 100 seconds, then terminate the execution of V3 and continue running V1 and V2 for 100 seconds. In this scenario, we assume that V1 and V2 are simply executed by default with no user parameters with the expectation that they deliver similar performance. In addition, we would like V3 to have twice the performance of V1 and V2.

Using the baseline performance measurements, we first describe what the expected ideal performance should be for this scenario. Since a single SLIC-Video player consumes nearly the entire machine, it is not possible to execute two video players at 30 fps. Given the baseline processing requirements for this application, it would be possible for each video player to sometimes display every frame and sometimes display every other frame, but it would be better for each video player to maintain a more constant time delta between the frames it displays. As a result, we would ideally expect that during the first 100 seconds of execution, V1 and V2 would each reduce their frame rate by skipping half of their respective frames and displaying the other half. Note that this does not require consuming 100% of the CPU. Ideally, this will result in the a 66.73 ms time delta between displayed frames for each application. In addition, since each digitizer still captures frames at 29.97

fps and its respective video player application processes the frame with the earliest timestamp that is stored in the digitizer's buffers, the delay between frame arrival and display should be 100.10 ms. By the same token when V3 begins, we would expect V1 and V2 to reduce their frame rate further, displaying only 25% of their frames, with delay and time delta of 166.83 ms and 133.47 ms, respectively. V3 should be able to display 50% of its frames, with delay and time delta of 100.10 ms and 66.73 ms, respectively. Upon termination of V3, we would ideally expect that the performance of V1 and V2 would be the same as during the first 100 seconds of their execution. In all cases, there should ideally be zero variation in the time delay between frame arrival and display and the time delta between frame displays.

We compare the ideal CPU allocations and application performance results with the actual CPU allocations and application performance results obtained when running the mix of video players on different schedulers. We considered five scheduler scenarios, which are discussed in further detail in Sections 6.3.4 through 6.3.7:

- UNIX TS: The three video applications are run using the TS scheduling class in UNIX SVR4, as discussed in Section 6.3.4.

- UNIX RT: The three video applications are run using the RT scheduling class in UNIX SVR4, as discussed in Section 6.3.5.

- UNIX TS2: The three video applications are again run as in the UNIX TS case, but the applications are modified to do a better job of synchronizing the video frame processing with the arrival of video frames. The modifications are discussed in detail in Section 6.3.6.

- SMART: The three video applications are run using SMART, as discussed in Section 6.3.7.

The CPU allocations obtained on the different schedulers are shown in Figure 6-10. The application performance results obtained on different schedulers are shown in Figures 6-11 through 6-15. Figure 6-11 shows the percentage of frames displayed. Figures 6-12 and 6-13 show the average and standard deviation in the delay between the arrival and display of each frame. Figures 6-14 and 6-15 show the average and standard deviation in the time delta between frames being displayed.

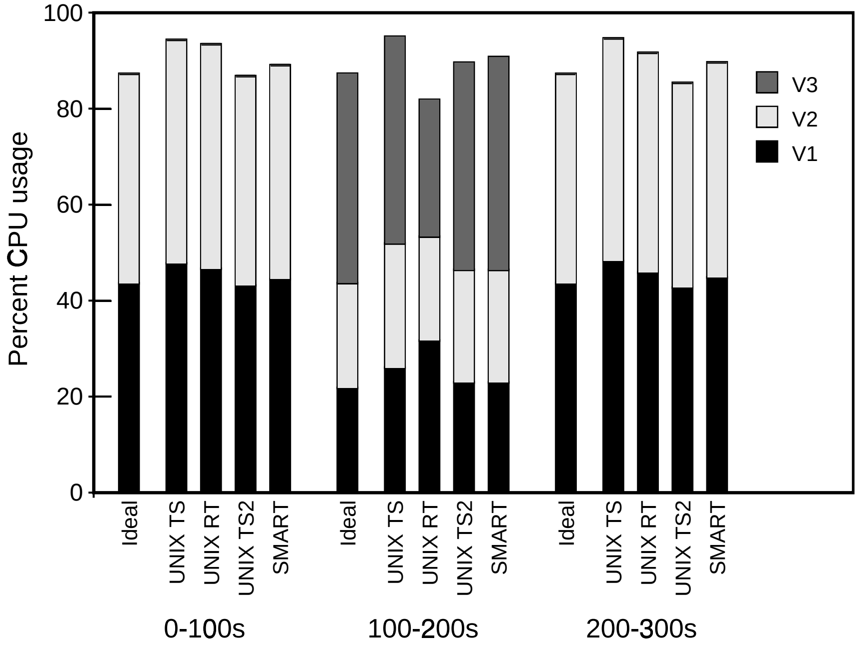


Figure 6-10. CPU allocations obtained by video players V1, V2, and V3 on different schedulers during three different time intervals

### 6.3.4. UNIX SVR4 Time-sharing Performance

We ran the experimental scenario described in Section 6.3.3 under standard UNIX TS scheduling. To give V3 roughly twice the performance of V1 and V2 under UNIX TS, extensive trial and error was required to find a suitable “nice” setting for V3 to bias the allocation of resources in accordance with the proportions desired. The nice setting for V3 was +15. The “UNIX TS” bars in Figure 6-10 show the resulting CPU allocations for this experiment. Despite extensive trial and error, there is a noticeable difference between the ideal CPU allocations and the CPU allocations obtained under UNIX TS for all of the video applications.

The second bar in Figures 6-12 and 6-14 show the average delay and time delta measurements for the video applications running under UNIX TS, and Figures 6-13 and 6-15 show the standard deviations in those measurements, respectively. While the average delay and average time delta measurements were quite acceptable, the standard deviation in those

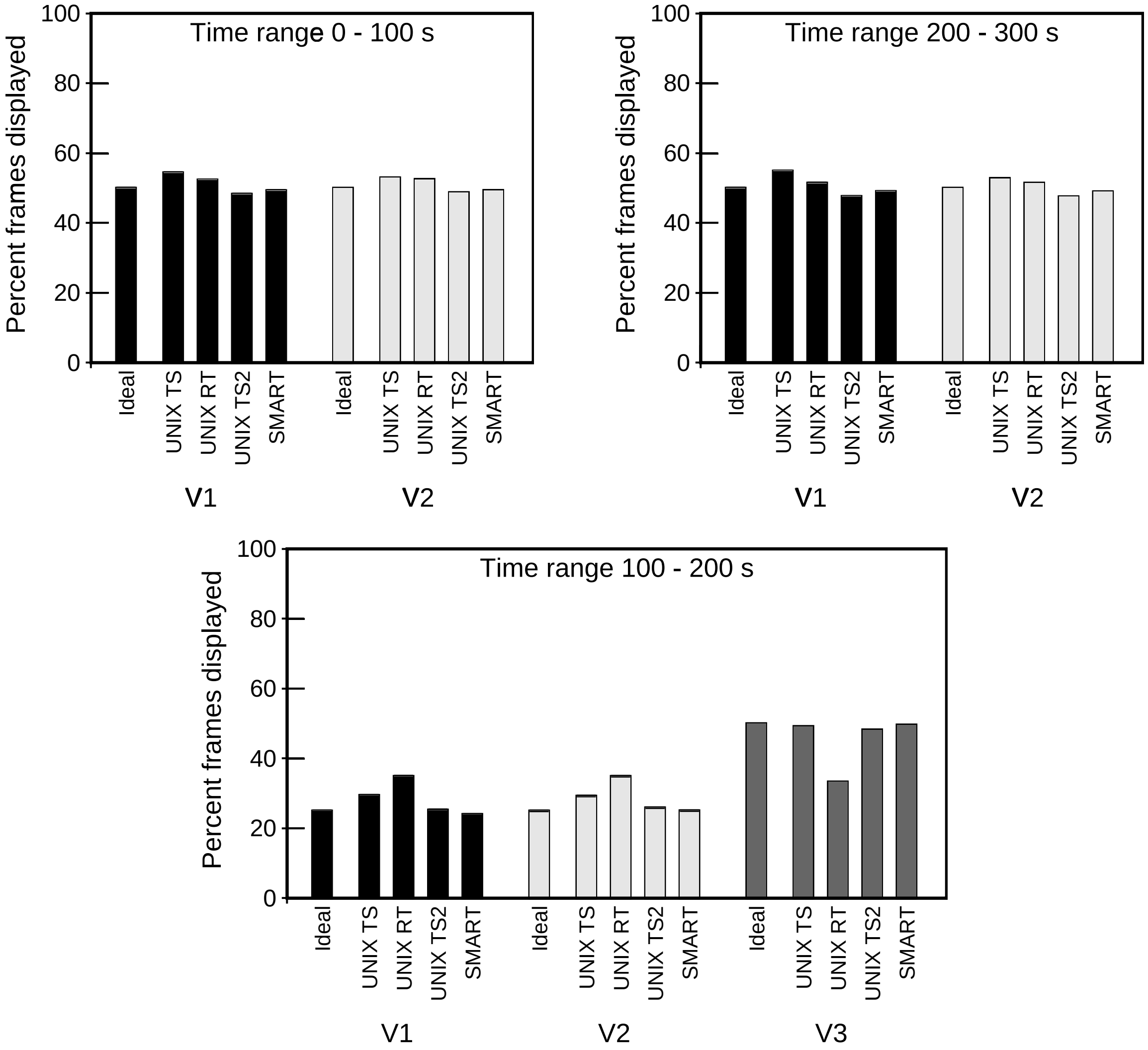


Figure 6-11. Percentage of frames displayed by video players V1, V2, and V3 when running on different schedulers during three different time intervals

measurements was not. During the first 100 seconds of execution with just V1 and V2 running, the standard deviation in the delay between frame arrival and frame display for V1 and V2 was more than 45 ms, and ballooned to more than 100 ms with V3 also running. The standard deviation in the time delta between frame displays with just V1 and V2 running was more than 50 ms, and grew to more than 100 ms with V3 also running. The performance of V3, while better than V1 or V2 during the same time interval, exhibited a large amount of video jitter as well. Its standard deviation in the delay between frame arrival and frame display, as well as its standard deviation in the time delta between frame displays, was over 65 ms, nearly two frame times of variation. The performance is far from ideal.

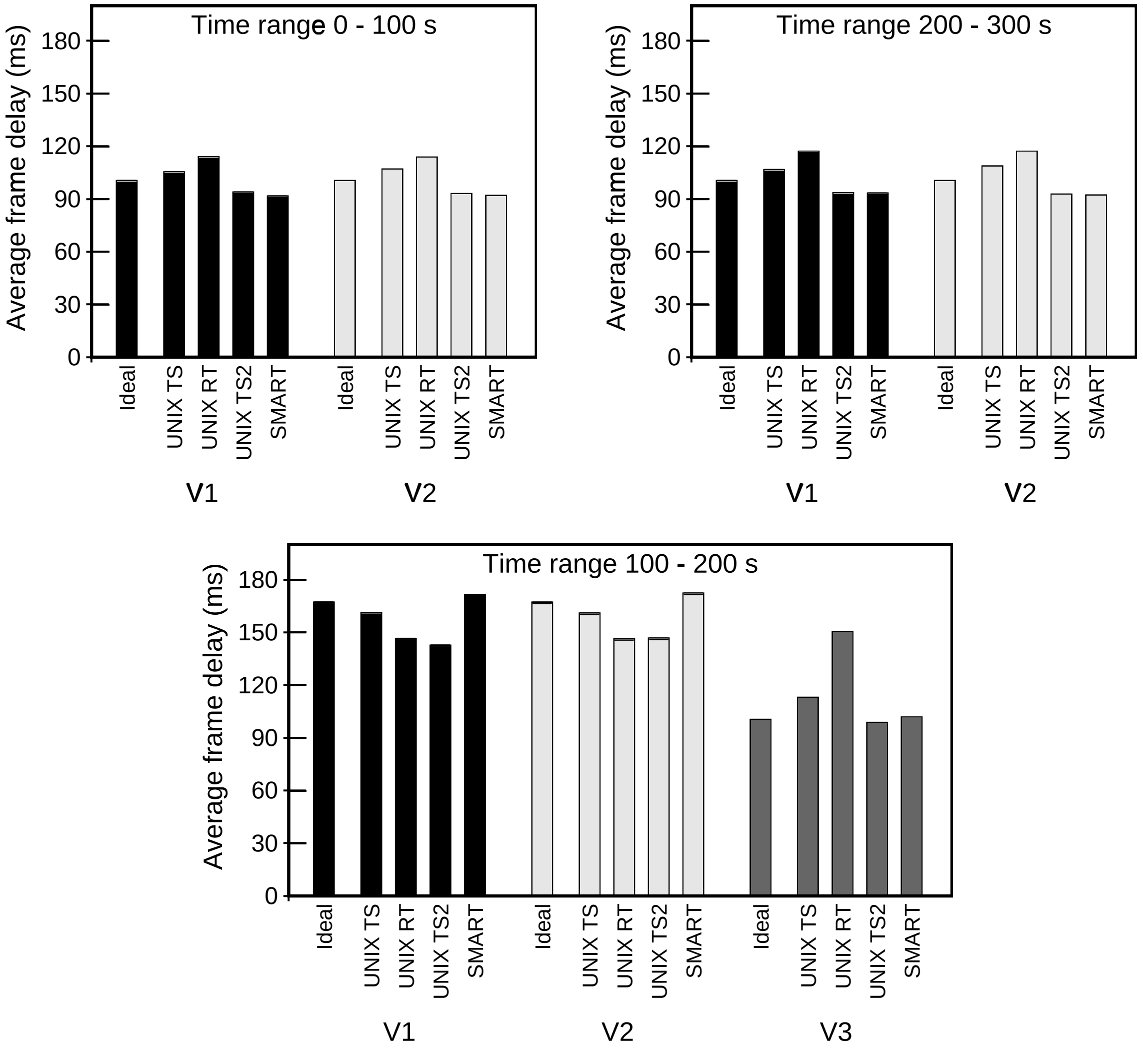


Figure 6-12. Average frame delay for video players V1, V2, and V3 when running on different schedulers during three different time intervals

### 6.3.5. UNIX SVR4 Real-time Performance

We ran the same experimental scenario of three video players under standard UNIX RT scheduling. V1 and V2 are both assigned the same default priority, while V3 is assigned a higher priority than either V1 or V2. The CPU allocations and application measurements for this experiment are contrasted with the results under UNIX TS in Figures 6-10 through 6-15.

While the standard deviations in the quality metrics are better than those under UNIX TS, UNIX RT suffers from two major problems. The first problem is evident by the

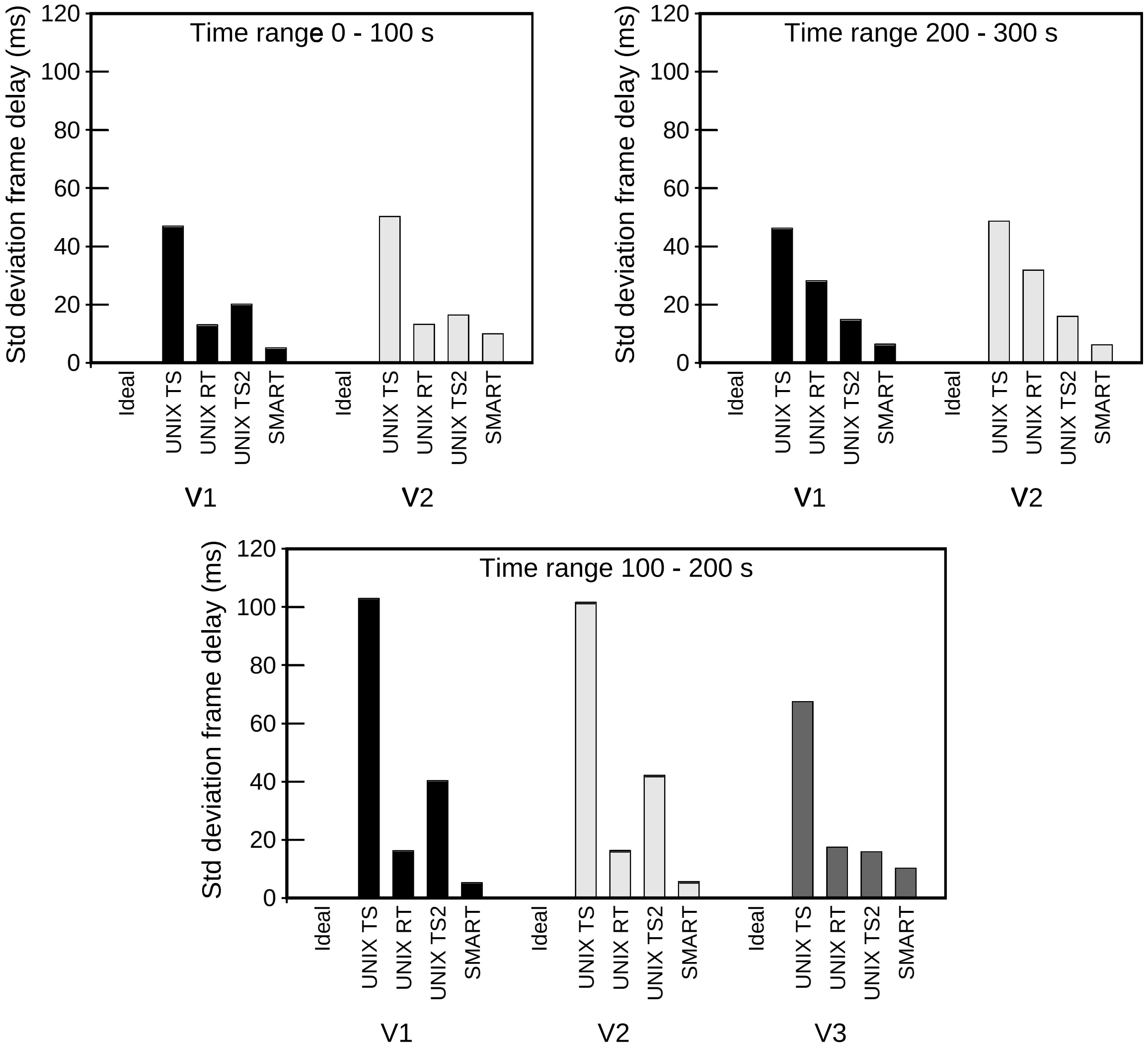


Figure 6-13. Standard deviation in the frame delay for video players V1, V2, and V3 when running on different schedulers during three different time intervals

performance measurements on V3. Since V3 has a higher priority than either V1 or V2, its performance should be better that of V1 or V2. However, the performance of V3 is actually somewhat worse than either of the other video players. The problem is that the signaling mechanism used to inform the user-level application of the arrival of a frame is a system-level function that therefore executes at a system-level priority. In UNIX SVR4, processes that are scheduled by the real-time scheduler are given higher priority than even system functions. The result is that the signal mechanism does not get to execute until all of the real-time video players finish processing their respective frames and block waiting to be informed of the arrival of a new frame to be processed. This serializes the execution of all

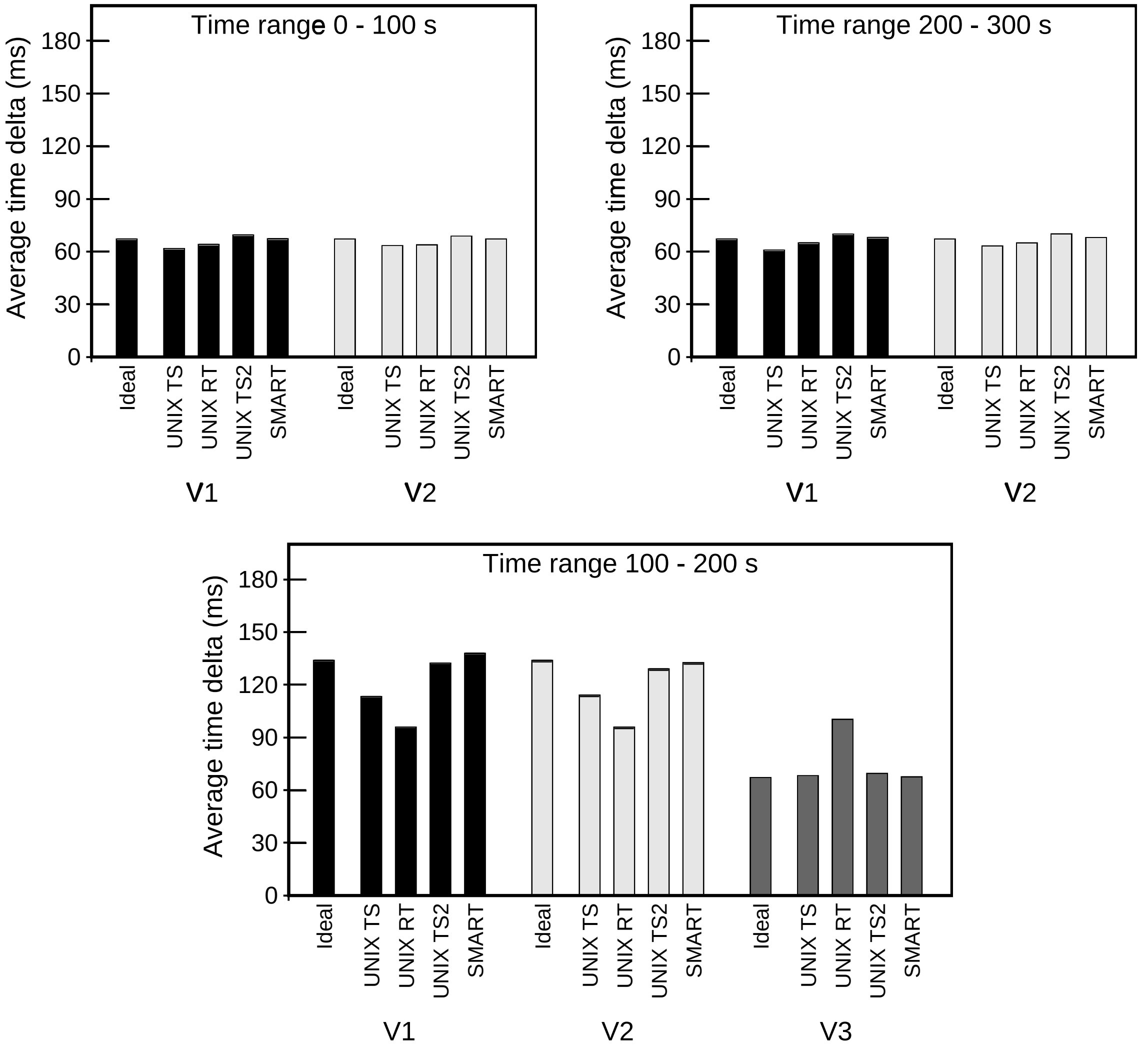


Figure 6-14. Average time delta between frames for video players V1, V2, and V3 when running on different schedulers during three different time intervals

of the video applications, irrespective of their assigned priority. The result for the end user is a complete lack of control in biasing the allocation of resources according to his preferences.

A more fundamental problem with running video under UNIX RT is that because the video applications are given the highest priority, they are able to take over the machine and starve out even the processing required to allow the system to accept user input. The result is that the user is unable to regain control of the system without restarting the system. Clearly UNIX RT is an unacceptable solution for multimedia applications.

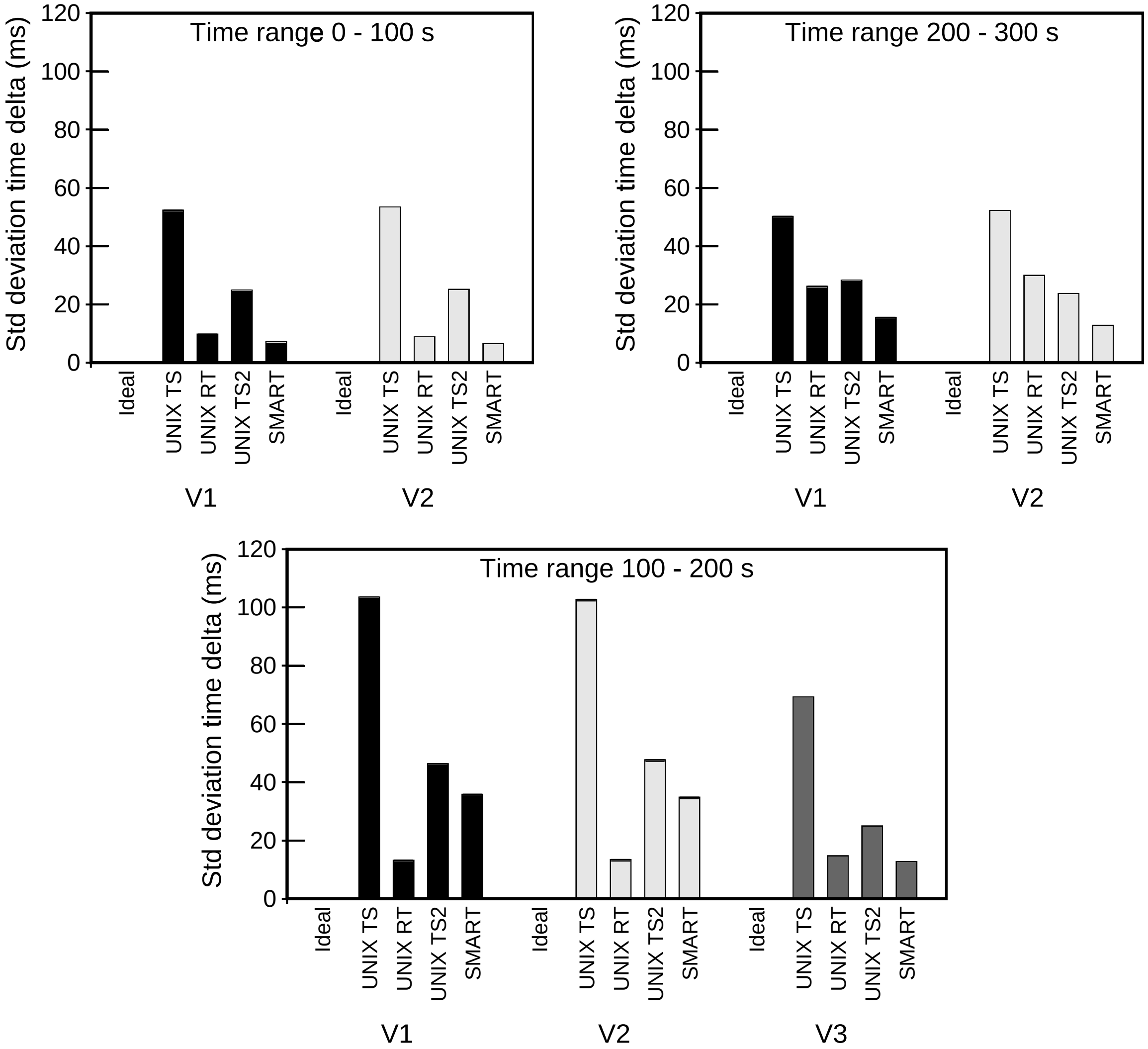


Figure 6-15. Standard deviation in the time delta between frames for video players V1, V2, and V3 when running on different schedulers during three different time intervals

### 6.3.6. Managing Time in UNIX SVR4 Time-sharing

The SLIC-Video production code used in the previous experiments does not explicitly account for the frame arrival times, nor does it explicitly attempt to adjust its rate of execution when many of the frames cannot be processed. Instead, it simply processes video frames as fast as possible. In particular, when the application finishes processing its current frame, if another frame has already arrived, the application simply processes the new frame immediately irrespective of when the frame arrived. This results in a substantial variance in the time delay between the arrival of the frame and its display. Rather than discarding frames that cannot be processed in time at a regularly spaced interval, the application haphazardly attempts to render whatever frames it can, implicitly discarding those frames cannot be processed before they are overwritten by the hardware.

In an attempt to address these problems and improve the performance under UNIX TS, the video application was modified to account for the frame arrival times in determining which frames it should render and when it should render each of those frames. The application selects a time delay in which to render its video frames. It measures the amount of wall clock time that elapses during the processing of each frame. Then, it uses an exponential average of the elapsed wall clock time of previously displayed frames as an estimate of how long it will take to process the current frame. This estimate is used in conjunction with the frame arrival time to determine if the given frame can be displayed on time. If the video player is ready to display its frame early, then it delays until the appropriate time; but if it is late, it discards its current frame. The application defines early and late as more than 16.68 ms (half of the time delta between arriving frames) early or late with respect to the selected time delay.

The selected time delay is used by the application to determine which frames to discard. The application will try to render 1 out of *N* frames, where *N* is the ratio of the selected time delay over the time delta between arriving frames. The application attempts to change its discard rate based on the percentage of frames that are rendered on time. If a large percentage of the frames rendered are late, the application will reduce its frame rate and increase its selected time delay accordingly. If the frames are all being rendered on time, the application will increase its frame rate and reduce its selected time delay to improve its quality of service. Note that the burden of these application modifications is placed squarely on the application developer; no assistance is provided by the scheduler.

We ran the same experimental scenario of three video players under standard UNIX TS with the above mentioned code modifications. The CPU allocations and application results are shown in Figures 6-10 through 6-15 as “UNIX TS2”. We see that the performance is better than UNIX TS without explicit time management in the application, and does not have the pathological behavior found with UNIX RT. However, the standard deviation in time delay for V1 and V2 while V3 is running is still more than 40 ms, which is far beyond the modest 16.83 ms threshold of timeliness used by the application. This is the result of two problems. One is that the scheduler, having no knowledge of the timing requirements of the application, does not allocate resources to each application at the right time. The other problem is that the application has a difficult time of selecting the best time delay and frame rate to use for the given loading condition. Without scheduler information, it must guess at when the load changes based on its own estimates of system load. The result is that its selected time delay and frame rate oscillate back and forth due to inaccurate knowledge of the allocation of processing time that the scheduler will give the application under the given system load. Guessing is not good enough.

### 6.3.7. SMART UNIX SVR4 Performance

To enable the SLIC-Video application to take advantage of SMART, three simple modifications were made to the code described in Section 6.3.6. First, rather than having the application rely on its own estimates of whether or not a frame is late and should be discarded, the application sets a time constraint that informs the scheduler of its deadline and cpu-estimate. The deadline is set to be 16.68 ms after the selected time delay. The cpu-estimate is calculated in the same manner as the average elapsed wall clock time: the application measures the execution time required for each frame and then uses an exponential average of the execution times of previously displayed frames as the cpu-estimate.

Second, upon setting the given time constraint, the application sets its notify-time equal to zero, thereby requesting the scheduler to notify the application right away if early estimates predict that the time constraint cannot be met. When a notification is sent to the application, the application’s notify-handler simply records the fact that the notification has been received. If the notification is received by the time the application begins the computation to process and display the respective video frame, the frame is discarded; otherwise, the application simply allows the frame be displayed late.

Third, rather than having to guess what the system loading condition is at any given moment, the application obtains its availability from the scheduler. It reduces its frame rate if frames cannot be completed on time and the required computational rate to process frames on time at the current frame rate is greater than its allocation rate. It increases its frame rate if the availability indicates that the consumption rate is less than its allocation rate.

We ran the same experimental scenario of three video players under SMART, taking advantage of its real-time API. The CPU allocation and application results of this experiment are shown in Figures 6-10 through 6-15, respectively. Not only does SMART effectively allocate CPU time in accordance with the user preferences for the experiment, SMART provides application results that are closest to the ideal performance figures.

In particular, SMART provides the smallest variation of any scheduler in the delay between frame arrival and frame display. The delay is well under 10 ms for all of the video players. Discounting the UNIX RT scheduler which ignores the user preferences, SMART also gives the smallest variation of any scheduler in the time delta between frames. The superior performance obtained by using the SMART interface can be attributed to two factors. One factor is that the scheduler accounts for the time constraints of the applications in managing resources. The second factor is that the application is able to adjust its frame rate more effectively because the SMART interface allows it to obtain availability information from the scheduler.

## 6.4. Multimedia, Interactive, and Batch Applications

While many multimedia application studies focus exclusively on audio or video applications, multimedia encompasses a much broader range of activities. In addition, it is important to realize that audio and video applications must co-exist with conventional interactive and batch applications in a general-purpose computing environment. We believe it is important to understand the interactions of these different classes of applications and provide good performance for all classes of applications.

To evaluate SMART in this context, we have conducted experiments on an application workload with a wide range of classes of applications in a fully-functional workstation environment. We describe two sets of experiments with a mix of real-time, interactive and batch applications executing in a workstation environment. The first experiment compares SMART with two existing schedulers: the UNIX SVR4 scheduler, both real-time (UNIX RT) and time-sharing (UNIX TS) policies, and a WFQ processor scheduler. These schedulers were chosen as a basis of comparison because of their common use in practice and research. UNIX SVR4 is a common basis of workstation operating systems used in practice, and WFQ is a popular scheduling technique that has been the basis of much recent scheduling research. The second experiment demonstrates the ability of SMART to provide the user with predictable resource allocation controls, adapt to dynamic changes in the workload, and deliver expected behavior when the system is not overloaded.

Three applications were used to represent batch, interactive and real-time computations:

- *Dhrystone* (batch) — This is the Dhrystone benchmark (Version 1.1), a synthetic benchmark that measures CPU integer performance.

- *Typing* (interactive) — This application emulates a user typing to a text editor by receiving a series of characters from a serial input line and using the X window server [60] to display them to the frame buffer. To enable a realistic and repeatable sequence of typed keystrokes for interactive applications, a hardware keyboard simulator was constructed and attached via a serial line to the testbed workstation. This device is capable of recording a sequence of keyboard inputs, and then replaying the sequence with the same timing characteristics.

- *Integrated Media Streams Player* (real-time) — The Integrated Media Streams (IMS) Player from Sun Microsystems Laboratories is a timestamp-based system capable of playing synchronized audio and video streams. As described in Section 3.5, the application was developed and tuned for the UNIX SVR4 time-sharing scheduler in the Solaris operating system. For the experiment with the SMART scheduler, we have inserted additional system calls to the application to take advantage of the features provided by SMART. The details of the modifications were described in Section 3.5. We use this application in two different modes:

*News* (real-time) — This application displays synchronized audio and video streams from local storage. Each media stream flows under the direction of an independent thread of control. The audio and video threads communicate through a shared memory region and use timestamps to synchronize the display of the media streams. The video input stream contains frames at 320x240 pixel resolution in JPEG compressed format at roughly 15 frames/second, though there is some variation in the time between successive timestamps. The audio input stream contains standard 8-bit μ-law monaural samples. The captured data is from a satellite news network. As described in Section 3.5, the audio and video streams were defined by the application to be synchronized if the respective video frame was displayed within 20 ms of the corresponding block of audio samples. A video frame is considered too early or too late if it is displayed more than 20 ms early or late with respect to the audio.

*Entertain* (real-time) — This application processes video from local storage. The video input stream contains frames at 320x240 pixel resolution in JPEG compressed format at roughly 15 frames/second. The application scales and displays the video at 640x480 pixel resolution. The captured data contains a mix of television programming, including sitcom clips and commercials. The video stream is not displayed in synchrony with any audio, so the timestamps on the video input stream determine when to display each frame. A video frame is considered too early or too late if it is displayed more than 20 ms early or late with respect to its associated timestamp.

### 6.4.1. Application Characteristics and Quality Metrics

Representing different classes of applications, *Typing*, *Dhrystone*, *News* and *Entertain* have very different characteristics and measures of quality. For example, we care about the response time for interactive activities, the throughput of batch activities and the number of deadlines met in real-time activities. Before discussing how a combination of these applications executes on different schedulers, this section describes how we measure the quality of each of the different applications, and how each would perform if it were to run on its own.

Table 6-2 shows the execution time of each application on an otherwise quiescent system using the UNIX SVR4 scheduler, measured over a time period of 300 seconds. We note that there is no significant difference between the performance of different schedulers when running only one application. The execution times include user time and system time spent on behalf of an application. The *Dhrystone* batch application can run whenever the processor is available and can thus fully utilize the processor. The execution of other system functions (fsflush, window system, etc.) takes less than 1% of the CPU time. The measurements on the real-time applications are taken every frame, and those for *Typing* are taken every character. None of the real-time and interactive applications can take up the whole machine on its own, with both *News* audio and *Typing* taking hardly any time at all. The video for *News* takes up 42% of the CPU, whereas *Entertain*, which displays scaled video, takes up almost 60% of the processor time.

| Name | Basis of Measurement | No. of Measurements | CPU Time Avg. | CPU Time Std. Dev. | % CPU Avg. |
|---|---|---|---|---|---|
| News audio | per segment | 4700 | 1.54 ms | 0.79 ms | 2.42% |
| News video | per frame | 4481 | 28.35 ms | 2.19 ms | 42.34% |
| Entertain | per frame | 4487 | 39.16 ms | 2.71 ms | 58.55% |
| Typing | per character | 1314 | 1.96 ms | 0.17 ms | 0.86% |
| Dhrystone | per execution | 1 | 298.73 s | N/A | 99.63% |

Table 6-2. Standalone execution times of applications

For each application, the quality of metric is different. For *Typing*, it is desirable to minimize the time between user input and system response to a level that is faster than what a human can readily detect. This means that for simple activities such as typing, cursor motion, or mouse selection, system response time should be less than 50-150 ms [62]. As such, we measured the *Typing* character latency and determine the percentage of characters processed with latency less than 50 ms, with latency between 50-150 ms, and with latency greater than 150 ms. For *News* audio, it is desirable not to have any artifacts in audio output. As such, we measured the number of *News* audio samples dropped. For *News* video and *Entertain*, it is desirable to minimize the difference between the desired display time and the actual display time, while maximizing the number of frames that are displayed within their time constraints. As such, we measured the percentage of *News* and *Entertain* video frames that were displayed on time (displayed within 20 ms of the desired time), displayed early, displayed late, and the percentage of frames dropped not displayed. Finally, for batch applications such as *Dhrystone*, it is desirable to maximize the processing time devoted to the application to ensure as rapid forward progress as possible. As such, we simply measured the CPU time *Dhrystone* accumulated. To establish a baseline performance, Table 6-3 shows the performance of each application when it was executed on its own.

| Name | Quality Metric | On Time | Early | Late | Dropped | Avg. | Std. Dev. |
|---|---|---|---|---|---|---|---|
| News audio | Number of audio dropouts | 100.00% | 0.00% | 0.00% | 0.00% | 0 | 0 |
| News video | Actual display time minus desired display time | 99.75% | 0.09% | 0.13% | 0.02% | 1.50 ms | 2.54 ms |
| Entertain | Actual display time minus desired display time | 99.58% | 0.22% | 0.13% | 0.07% | 1.95 ms | 3.61 ms |
| Typing | Delay from character input to character display | 100.00% | N/A | 0% | N/A | 26.40 ms | 4.12 ms |
| Dhrystone | Accumulated CPU time | N/A | N/A | N/A | N/A | 298.73 s | N/A |

Table 6-3. Standalone application quality metric performance

While measurements of accumulated CPU time are straightforward, we note that several steps were taken to minimize and quantify any error in measuring audio and video performance as well as interactive performance. For *News* and *Entertain*, the measurements reported here are performed by the respective applications themselves during execution. We also quantified the error of these internal measurements by using a hardware device to externally measure the actual user perceived video display and audio display times [61]. External versus internal measurements differed by less than 10 ms. The difference is due to the refresh time of the frame buffer. For *Typing*, we measured the end-to-end character latency from the arrival of the character to the system in the input device driver, through the processing of the character by the application, until the actual display of the character by the X window system character display routine.

### 6.4.2. Scheduler Characteristics

To provide a characterization of scheduling overhead, we measured the context switch times for the UNIX SVR4, WFQ, and SMART schedulers. Average context switch times for UNIX SVR4, WFQ, and SMART are 27 μs, 42 μs, and 47 μs, respectively. These measurements were obtained running the mixes of applications described earlier in this section.

Similar results were obtained when we increased the number of real-time multimedia applications in the mix up to 15, at which point no further multimedia applications could be run due to there being no more memory to allocate to the applications.

The UNIX SVR4 context switch time essentially measures the context switch overhead for a scheduler that takes almost no time to decide what activity it needs to execute. The scheduler simply selects the highest priority activity to execute, with all activities already sorted in priority order. Note that this measure does not account for the periodic processing done by the UNIX SVR4 timesharing policy to adjust the priority levels of all activities. Such periodic processing is not required by WFQ or SMART, which makes the comparison of overhead based on context switch times more favorable for UNIX SVR4. Nevertheless, as activities are typically scheduled for time quanta of several milliseconds, the measured context switch times for all of the schedulers were not found to have a significant impact on application performance.

For SMART, we also measured the cost to an application of assigning scheduling parameters such as time constraints or reading back scheduling information. The cost of assigning scheduling parameters to an activity is 20 μs while the cost of reading the scheduling information for an activity is only 10 μs. The small overhead easily allows application developers to program with time constraints at a fine granularity without much penalty to application performance.

### 6.4.3. Comparison of Default Scheduler Behavior

Our first experiment is simply to run all four applications (*News*, *Entertain*, *Typing*, and *Dhrystone*) with the default user parameters for each of the schedulers:

- UNIX RT: The real-time *News* and *Entertain* applications are put in the real-time class, leaving *Typing* and *Dhrystone* in the time-sharing class.
- UNIX TS: All the applications are run in time-sharing mode. (We also experimented with putting *Typing* in the interactive application class and obtained slightly worse performance.)

- WFQ: All the applications are run with equal share.

- SMART: All the applications are run with equal share and equal priority.

Because of their computational requirements, the execution of these applications results in the system being overloaded. In fact, the *News* video and the *Entertain* applications alone will fully occupy the machine. Both the *Typing* and *News* audio applications hardly use any CPU time, taking up a total of only 3-4% of the CPU time. It is thus desirable for the scheduler to deliver short latency on the former application and meet all the deadlines on the latter application. With the default user parameters in UNIX TS, WFQ, and SMART, we expect the remainder of the computation time to be distributed evenly between *News* video, *Entertain*, and *Dhrystone*. Even with an ideal scheduler, we expect the percentages of the frames dropped to be 25% and 45% for *News* video and *Entertain*, respectively.

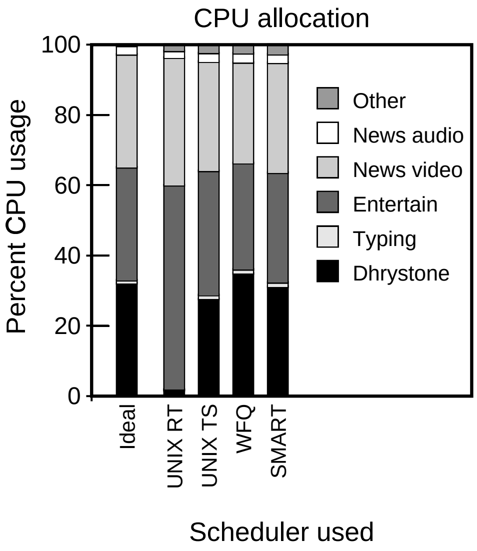


Figure 6-16. CPU allocations obtained by applications when run with default behavior on different schedulers

Figure 6-16 presents the CPU allocation across different applications by different schedulers. It includes the percentage of the CPU used for executing other system functions such as the window system (labeled *Other*). The figure also includes the expected result of an ideal scheduler for comparison purposes. For the real-time applications, Figures 6-17 and 6-18 show the percentage of media units that are displayed on-time, early, late, or dropped.

For the interactive *Typing* application, Figure 6-19 shows the number of characters that take less than 50 ms to display, take 50-150 ms to display, and take longer than 150 ms to display. Figures 6-20 through 6-23 present more detail by showing the distributions of the data points. We have also included the measurements for each of the applications running by itself (labeled *Standalone*) in the respective figures. We observe that every scheduler handles the *News* audio application well with no audio dropouts. Thus we will only concentrate on discussing the quality of the rest of the applications.

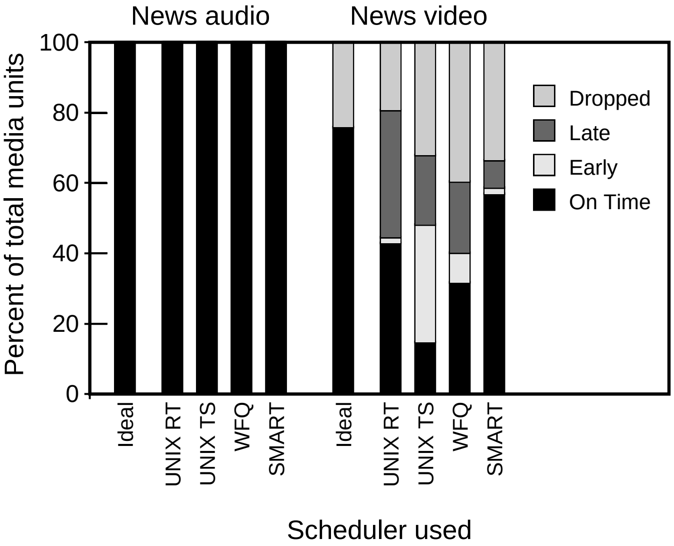


Figure 6-17. *News* application performance on different schedulers as measured by percentage of audio and video samples delivered on time

Unlike the other schedulers, the UNIX RT scheduler gives higher priority to applications in the real-time class. It devotes most of the CPU time to the video applications, and thus drops the least number of frames. (Nevertheless, SMART is able to deliver more on-time frames than UNIX RT for the *News* video, while using less resources.) Unfortunately, UNIX RT runs the real-time applications almost to the exclusion of conventional applications. *Dhrystone* gets only 1.6% of the CPU time. More disturbingly, the interactive *Typing* application does not get even the little processing time requested, receiving only 0.24% of the CPU time. Only 635 out of the 1314 characters typed are even processed within the 300 second duration, and nearly all the characters processed have an unacceptable latency of greater than 150 ms. Note that putting *Typing* in the real-time class does not alleviate this problem as the system-level I/O processing required by the application is still not able to run, because system functions are run at a lower priority than real-time activities. Clearly, it is not acceptable to use the UNIX RT scheduler.

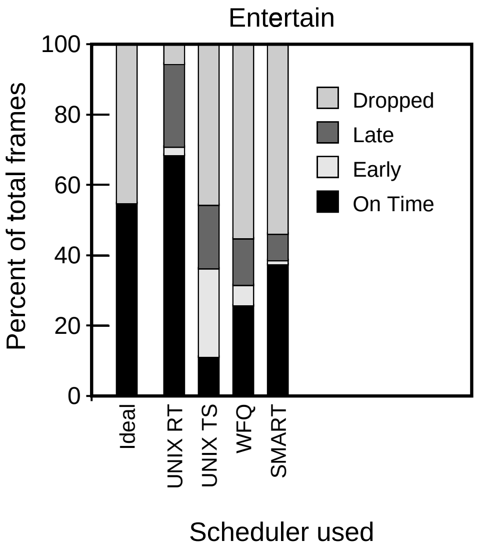


Figure 6-18. *Entertain* application performance on different schedulers as measured by percentage of video frames delivered on time

All the other schedulers spread the resources relatively evenly across the three demanding applications. The UNIX TS scheduler has less control over the resource distribution than WFQ and SMART, resulting in a slight bias towards *Entertain* over *Dhrystone*. The basic principles used to achieve fairness across applications are the same in WFQ and SMART. However, we observe that WFQ scheduler devotes slightly more (3.8%) CPU time to *Dhrystone* at the expense of *News* video. This effect can be attributed to the standard implementation of WFQ processor scheduling whereby the proportional share of the processor obtained by an activity is based only on the time that the activity is runnable and does not include any time that the activity is sleeping.

Since the video applications either process a frame or discard a frame altogether from the beginning, the number of video frames dropped is directly correlated with the amount of time devoted by the scheduler to the applications, regardless of the scheduler used. The difference in allocation accounts for the difference in the number of frames dropped between the schedulers. We found that in each instance the scheduler drops about 6-7% more frames than the ideal computed using average computation times and the scheduler's specific allocation for the application.

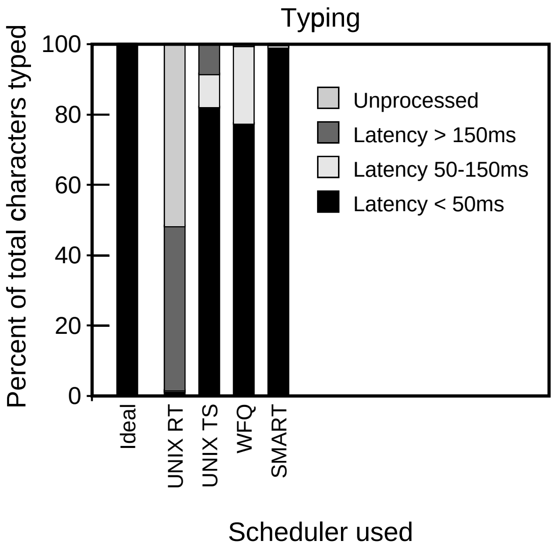


Figure 6-19. *Typing* application performance on different schedulers as measured by character latency

The schedulers are distinguished by their ability to meet the time constraints of those frames processed. SMART meets a significantly larger number of time constraints than the other schedulers, delivering over 250% more video frames on time than UNIX TS and over 60% more video frames on time than WFQ. SMART's effectiveness holds even for cases where it processes a larger total number of frames, as in the comparison with WFQ. Moreover, as shown in Figures 6-20 and 6-21, the late frames are handled soon after the deadlines, unlike the case with the other schedulers. As SMART delivers a more predictable behavior, the applications are better at determining how long to sleep to avoid displaying the frames too early. As a result, there is a relatively small number of early frames. It delivers on time 57% and 37% of the total number of frames in *News* video and *Entertain*, respectively. They represent, respectively, 86% and 81% of the frames displayed.

To understand the significance of the bias introduced to improve the real-time and interactive application performance, we have also performed the same experiment with all biases set to zero. The use of the bias is found to yield a 10% relative improvement on the scheduler's ability in delivering the *Entertain* frames on time.

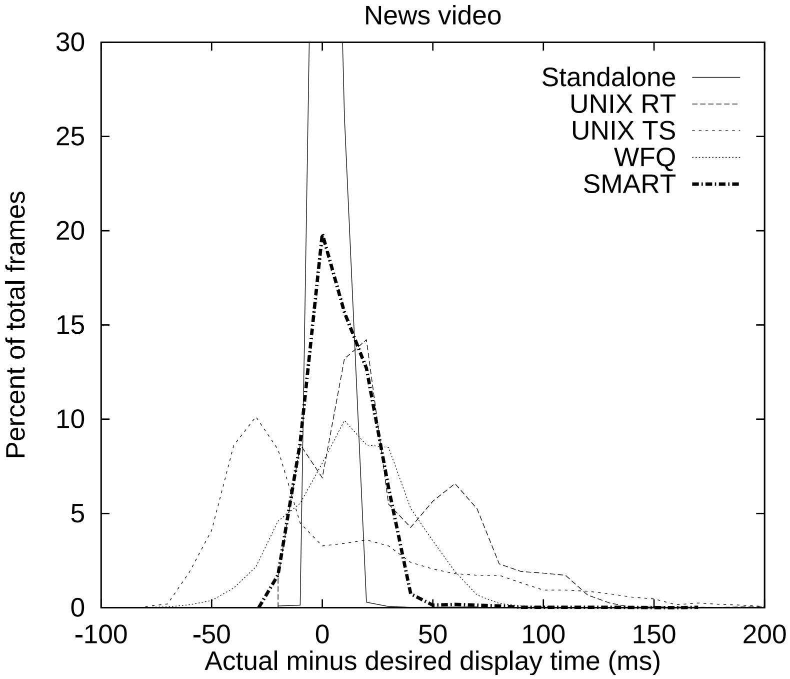


Figure 6-20. Distributions of frame display times for the *News* application on different schedulers

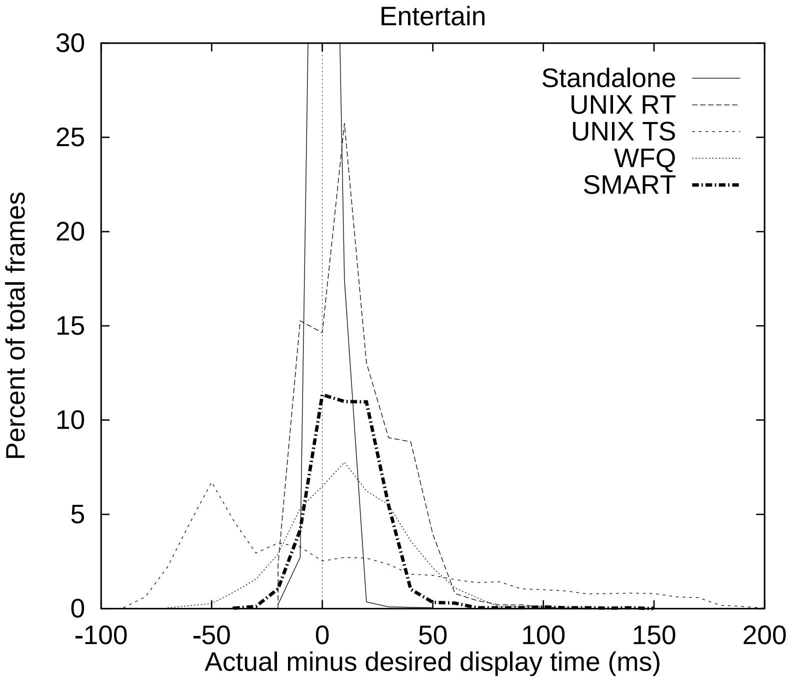


Figure 6-21. Distributions of frame display times for the *Entertain* application on different schedulers

In contrast, the WFQ delivers 32% and 26% of the total frames on time, which represents only 53% and 58% of the frames processed. There are many more late frames in the WFQ case than in SMART. The tardiness causes the applications to initiate the processing earlier, thus resulting in a correspondingly larger number of early frames. The UNIX TS performs even more poorly, delivering 15% and 11% of the total frames, representing only 22% and 21% of the frames processed. Some of the frames handled by UNIX TS are extremely late, causing many frames to be processed extremely early, resulting in a very large variance in display time across frames.

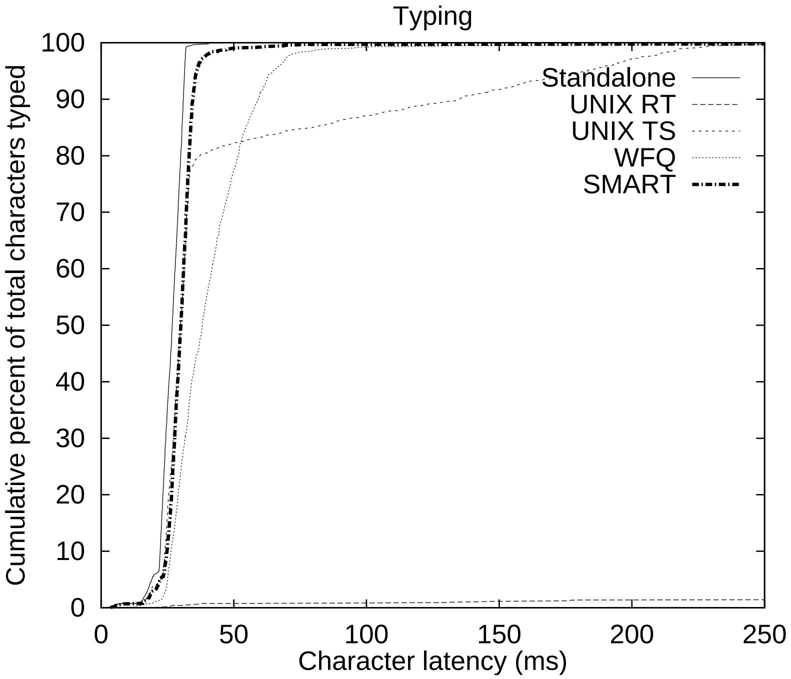


Figure 6-22. Distributions of character latency for the *Typing* application on different schedulers

Finally, as shown in Figure 6-22, SMART is superior to both SVRT-TS and WFQ in handling the *Typing* application. SMART has the least average and standard deviation in character latency and completes the most number of characters in less than 50 ms, the threshold of human detectable delay.

While both SMART and WFQ deliver acceptable interactive performance, *Typing* performs worse with WFQ because an activity does not accumulate any credit at all when it sleeps. We performed an experiment where the WFQ algorithm is modified to allow the blocked activity to accumulate limited credit just as it would when run on the SMART scheduler. The result is that *Typing* improves significantly, and the video application gets a fairer share of the resources. However, even though the number of dropped video frames is reduced slightly, the modified WFQ algorithm has roughly the same poor performance as before when it comes to delivering the frames on time.

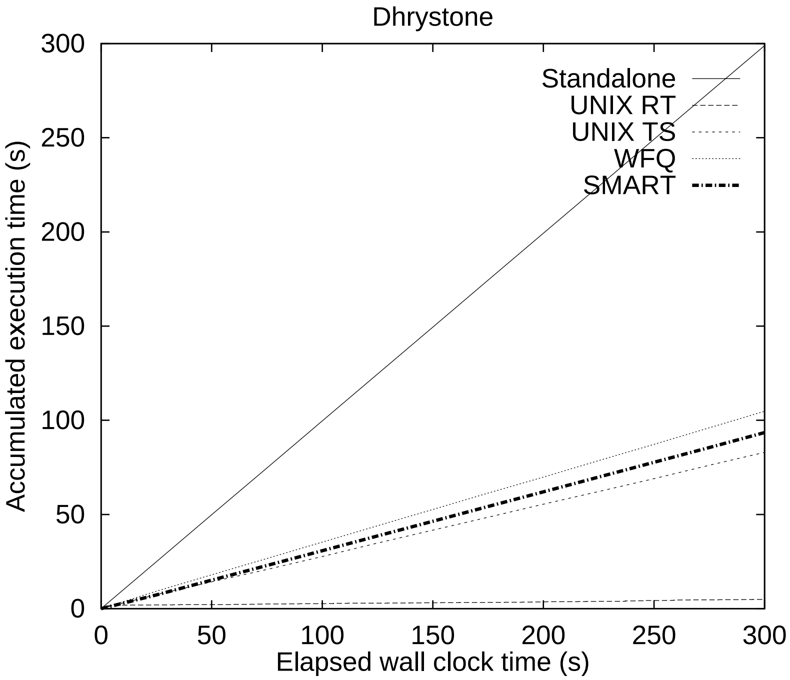


Figure 6-23. Cumulative execution time for the *Dhrystone* application on different schedulers

### 6.4.4. Adjusting the Allocation of Resources

Besides being effective for real-time applications, SMART has the ability to support arbitrary shares and priorities and to adapt to different system loads. We illustrate these features by running the same set of applications from before with different priority and share assignments under different system loads. In particular, *News* is given a higher priority than all the other applications, *Entertain* is given the default priority and twice as many shares as any other application, and all other applications are given the same default priority and share. This level of control afforded by SMART's priorities and shares is not possible with other schedulers. The experiment can be described in two phases:

- *Phase 1*: Run all the applications for the first 120 seconds of the experiment. *News* exits after the first 120 seconds of the experiment, resulting in a load change.

- *Phase 2*: Run the remaining applications for the remaining 180 seconds of the experiment.

Besides *News* and *Entertain,* the only other time-consuming application in the system is *Dhrystone*. Thus, in the first part of the experiment, *News* should be allowed to use as much of the processor as necessary to meet its resource requirements since it has a higher priority than all other applications. Since *News* audio uses less than 3% of the machine and *News* video uses only 42% of the machine on average, over half of the processor's time should remain available for running other applications. As *Typing* consumes very little processing time, it should be handled perfectly and almost all of the remaining computation time should be distributed between *Entertain* and *Dhrystone* in the ratio 2:1. The time allotted to *Entertain* can service at most 62% of the deadlines on average. When *News* finishes, however, *Entertain* is allowed to take up to 2/3 of the processor, which would allow the application to run at full rate. The system is persistently overloaded in Phase 1 of the experiment, and on average underloaded in Phase 2, though transient overloads may occur due to fluctuations in processing requirements.

Figures 6-24 through 6-27 show the CPU allocation and quality metrics of the different applications run under SMART as well as an ideal scheduler. The figures show that SMART's performance comes quite close to the ideal. First, it implements proportional sharing well in both underloaded and overloaded conditions. Second, SMART performs well for higher priority real-time applications and real-time applications requesting less than their fair share of resources. In the first phase of the computation, it provides perfect *News* audio performance, and delivers 97% of the frames of *News* video on time and meets 99% of the deadlines. In the second phase, SMART displays 98% of the *Entertain* frames on time and meets 99% of the deadlines. Third, SMART is able to adjust the rate of the application requesting more than its fair share, and can meet a reasonable number of its deadlines. In the first phase for *Entertain*, SMART drops only 5% more total number of frames than the ideal, which is calculated using average execution times and an allocation of 33% of the processor time. Finally, SMART provides excellent interactive response for *Typing* in both overloaded and underloaded conditions. 99% of the characters are displayed with a delay unnoticeable to typical users of less than 100 ms [8].

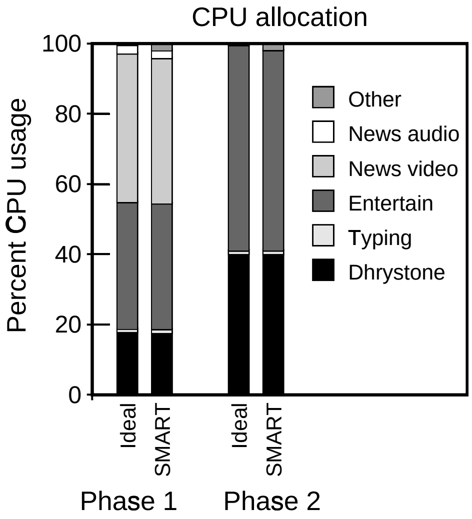


Figure 6-24. CPU allocations for different applications under a changing load when using SMART end user controls

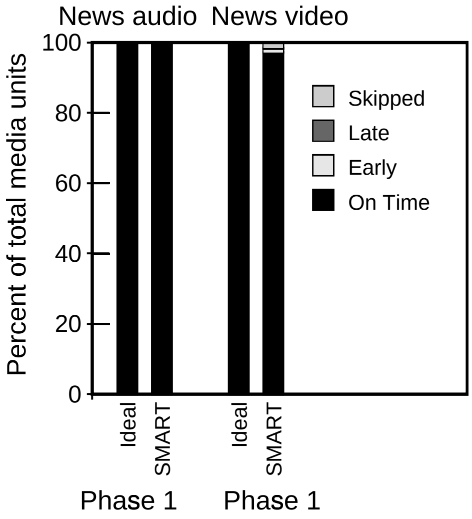


Figure 6-25. *News* application performance under a changing load when using SMART end user controls

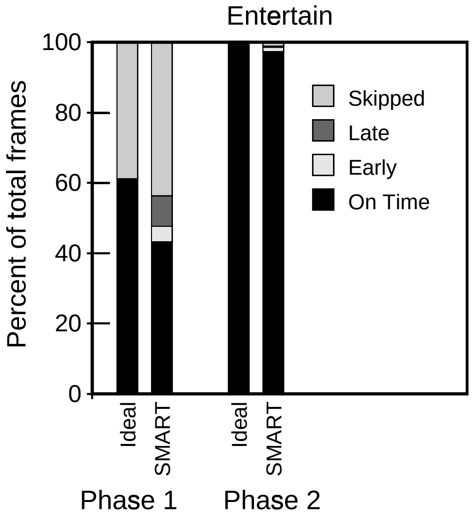


Figure 6-26. *Entertain* application performance under a changing load when using SMART end user controls

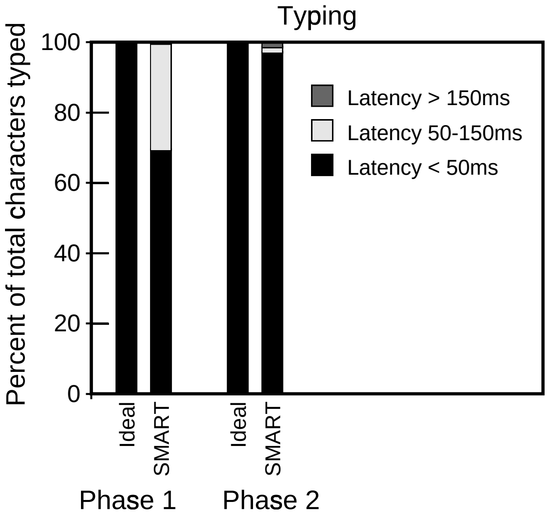


Figure 6-27. *Typing* application performance under a changing load when using SMART end user controls

### 6.4.5. Summary

Our experiments in the context of a full featured, commercial, general-purpose operating system show that SMART: (1) reduces the burden of writing adaptive real-time applications, (2) has the ability to cooperate with applications in managing resources to meet their dynamic time constraints, (3) provides resource sharing across both real-time and conventional applications, (4) delivers improved real-time and interactive performance over other schedulers without any need for users to reserve resources, adjust scheduling parameters, or know anything about application requirements, (5) provides flexible, predictable controls to allow users to bias the allocation of resources according to their preferences. SMART achieves this range of behavior by differentiating between the importance and urgency of real-time and conventional applications. This is done by integrating priorities and weighted fair queueing for importance, then using urgency to optimize the order in which tasks are serviced based on earliest-deadline scheduling. Our measured performance results demonstrate SMART's effectiveness over that of other schedulers in supporting multimedia applications in a realistic workstation environment.

# 7 Conclusion

We are at the dawn of an era in which many everyday applications will use multimedia. Tomorrow's multimedia applications will do much more than just playback pre-recorded audio and video; we expect that such applications will employ sophisticated image processing techniques and integrate complex computer graphics, all delivered with high interactivity and real-time response. However, multimedia applications have very different characteristics from the conventional non-real-time applications that populate desktop computers today. They are typically highly resource intensive, and often have dynamic and adaptive application-specific time constraints associated with their execution. To integrate these applications into the general-purpose computing environment, multitasking software environments must be able to support the demands of real-time multimedia applications in conjunction with the demands of existing conventional applications. To allow this combination of real-time and conventional activities to co-exist, general-purpose operating systems must effectively manage computing resources to meet the real-time demands of multimedia applications while still providing good performance for conventional interactive and batch applications. Operating systems have so far been unable to effectively manage resources to support this combination of real-time and conventional activities. To enable the wide-spread use of multimedia, operating systems must evolve beyond their current resource management limitations.

This dissertation represents a step towards enabling the wide-spread use of multimedia in general-purpose computing environments. In this dissertation, we have developed a processor scheduler that supports the co-existence of dynamic, adaptive real-time applications with conventional non-real-time applications. We have shown that this scheduler can be implemented in a commercial operating system and deliver significant performance improvements for both real-time and conventional applications over other schedulers used in research and practice. This dissertation makes the following contributions:

- We have conducted the first experimental studies that quantitatively evaluated UNIX SVR4's ability to support multimedia applications. UNIX SVR4 serves as a common basis of commercial workstation operating systems and claims to provide support for multimedia applications. We demonstrated that UNIX SVR4 processor scheduling is inadequate, resulting in pathological behaviors in which the video would freeze and the system would even stop accepting user input.

- We have created a new scheduler interface that enables users and applications to cooperate with the operating system in managing resources to support multimedia users and applications. This interface facilitates a greater flow of information among users, applications, and the operating system. It allows an operating system to account for both application and user information in managing resources, yet in no way imposes draconian demands on either application developers or end users for information they cannot or choose not to provide.

- We have developed a novel asynchronous notification mechanism to provide dynamic feedback to real-time applications to inform them if their time constraints cannot be met and enables applications to define their own policies for adapting their quality of service to the current system load. Possible adaptation policies that can take advantage of the notification mechanism include discarding a computation that will miss its deadline, progressively refining a computation until its deadline, continuing a computation after its deadline, and simply defining a new deadline for a computation.

- We have developed a scheduler interface and algorithm that is the first to support proportional share control at different priority levels across both real-time and conventional activities. These priority and proportional share controls give end users simple predictable controls that can be used to bias the allocation of resources according to their preferences.

- We have created a unified approach to scheduling real-time and conventional activities. We show that our SMART scheduler is the first to provide optimal performance for real-time applications when the system is underloaded while simultaneously providing proportional share control across real-time and conventional activities. Our scheduling algorithm is able to provide good combined real-time and conventional application performance even in the absence of admission control policies.

- We have introduced the notion of latency tolerance as a mechanism to improve the response time of interactive applications by adjusting the instantaneous allocation of resources that an application receives. However unlike multi-level feedback schedulers, our latency tolerance mechanism does not change the overall allocation of resources that an application receives.

- We have implemented a prototype of SMART in a commercial operating system environment. We have shown that is possible to implement SMART in such a way that provides effective support for multimedia applications while being completely backwards compatible with a UNIX SVR4 scheduling framework. Our implementation in fact supports all of the default scheduling classes in UNIX SVR4. We have demonstrated the robustness of our implementation in running real multimedia, interactive, and batch applications in a fully-functional workstation environment.

- We have demonstrated the effectiveness of our unified approach to scheduling by quantitatively comparing it with other schedulers used in research and practice. By measuring the performance of actual real-time multimedia, interactive, and batch applications in a fully-functional workstation environment, we show that SMART provides better performance and control than other schedulers. In fact, SMART can deliver almost a factor of two better performance than schedulers used in practice and research in meeting real-time requirements when the system is overloaded.

We have shown that SMART provides effective processor scheduling for real multimedia applications in a general-purpose computing environment.

## 7.1. Future Work

Effective uniprocessor scheduling is crucial for multimedia applications, but multiprocessor scheduling support for multimedia applications is becoming increasingly important. While some previous work has attempted to address the problem of real-time multiprocessor scheduling [14][19][21], little work has been done to address the problem of how to allow both real-time and conventional applications to share resources and co-exist together in a multiprocessor environment. Scheduling multimedia applications on multiprocessors poses challenges that do not arise in scheduling single processor systems. A single dispatch queue from which tasks are scheduled is sufficient for the uniprocessor case. However, experience with commercial operating systems suggests that scheduling multiple processors with a centralized dispatch queue is a synchronization bottleneck that can limit the scalability of multiprocessor systems [18][72]. If a dispatch queue should be associated with each processor, how should tasks be assigned to those dispatch queues in the first place? In particular, the operating system must effectively balance the load across multiple processors. As multimedia application workloads often have dynamically varying resource demands, the operating system must decide when to migrate tasks from one processor to another for load balancing. On the other hand, the operating system may want to reduce task migration through some form of cache affinity to reduce cache misses that occur when a task migrates among processors.

The SMART scheduler prototype implemented in the Solaris operating system is capable of scheduling activities in multiprocessor systems, but it does not currently address a number of multiprocessor scheduling issues such as scheduling real-time activities across multiple processors, load balancing among processors, and accounting for possible cache effects in determining which processor to use for executing a given activity. Nevertheless, our experience with SMART leads us to believe that the basic ideas in SMART can be extended to address these multiprocessor scheduling issues in supporting multimedia applications. In fact, we have begun experimenting with a new multiprocessor scheduler along these lines. Key features of our approach are: (1) decouples the assignment of which processor to use to run a given task (processor task assignment) from the scheduling of tasks already assigned to a processor (per processor scheduling), (2) accounts for cache effects in the processor task assignment by recognizing that conventional applications often have good cache locality but multimedia applications that stream large amounts of data often have poor cache locality, (3) explicitly accounts for application time constraints in both the processor task assignment and the per processor scheduling to make efficient use of resources in meeting real-time requirements, (4) provides flexible prioritized and proportional resource sharing across both real-time and conventional activities. In particular for our multiprocessor scheduler, we take advantage of SMART's flexible usage model and interface and employ the SMART scheduling algorithm for per processor scheduling.

While effective processor scheduling is crucial to support multimedia applications, processors are just one set of components in an overall system. Other resources that require effective resource management include I/O bandwidth, memory, networks, and the network/host interface. Meeting the demands of future multimedia applications will require coordinated resource management across all critical resources in the system. Providing resource management mechanisms and policies across multiple resources that effectively support adaptive and interactive multimedia applications remains a key challenge. We believe that the ideas discussed here for processor scheduling will serve as a basis for future work in addressing the larger problem of managing system-wide resources to support multimedia applications.

# Bibliography

[1] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall Inc., Englewood Cliffs, NJ, 1986.

[2] D. R. Bacher, “Content-based Indexing of Captioned Video”, S.B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994.

[3] V. Baiceanu, C. Cowan, D. McNamee, C. Pu, and J. Walpole, “Multimedia Applications Require Adaptive CPU Scheduling”, *Proceedings of the IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems*, Washington, DC, Dec. 1996.

[4] J. Barton and N. Bitar, “A Scalable Multi-discipline, Multi-processor Scheduling Framework for IRIX”, *Proceedings of Workshop on Job Scheduling for Parallel Processing*, Santa Barbara, CA, pp. 45-69, Apr. 1995.

[5] A. C. Bavier, A. B. Montz, and L. L. Peterson, “Predicting MPEG Execution Times”, *Proceedings of SIGMETRICS ‘98*, pp. 131-140, June 1998.

[6] J. C. R. Bennett and H. Zhang, “$WF^2Q$: Worst-case Fair Weighted Fair Queueing”, *IEEE INFOCOM ‘96*, San Francisco, CA, pp. 120-128, Mar. 1996.

[7] G. Bollella and K. Jeffay, “Support for Real-Time Computing Within General Purpose Operating Systems: Supporting Co-Resident Operating Systems”, *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Chicago, IL, pp. 4-14, May 1995.

[8] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*, L. Erlbaum Associates, Hillsdale, NJ, 1983.

[9] R. K. Clark, “Scheduling Dependent Real-Time Activities”, Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, Aug. 1990.

[10] G. Coulson, A. Campbell, P. Robin, G. Blair, M. Papathomas, and D. Hutchinson, “The Design of a QoS Controlled ATM Based Communications System in Chorus”, *IEEE JSAC*, 13(4), pp. 686-699, May 1995.

[11] H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.

[12] A. Demers, S. Keshav, and S. Shenker, “Analysis and Simulation of a Fair Queueing Algorithm”, *Proceedings of SIGCOMM ‘89*, pp. 1-12, Sept. 1989.

[13] M. Dertouzos, “Control Robotics: The Procedural Control of Physical Processors”, *Proceedings of the IFIP Congress*, Stockholm, Sweden, pp. 807-813, Aug. 1974.

[14] M. Dertouzos and A. Mok, “Multiprocessor On-line Scheduling of Hard-Real-Time Tasks”, *IEEE Transactions on Software Engineering*, 15(12), pp. 1497-1506, Dec. 1989.

[15] M. Dertouzos, “Creating the People’s Computer”, *MIT Technology Review*, Association of Alumni and Alumnae of the Massachusetts Institute of Technology, pp. 20-28, Apr. 1997.

[16] R. B. Essick, “An Event-based Fair Share Scheduler”, *Proceedings of the 1990 Winter USENIX Conference*, Washington, DC, pp. 147-161, Jan. 1990.

[17] S. Evans, K. Clarke, D. Singleton, and B. Smaalders, “Optimizing Unix Resource Scheduling for User Interaction”, *Proceedings of the 1993 Summer USENIX Conference*, Cincinnati, OH, pp. 205-218, June 1993.

[18] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, “Beyond Multiprocessing...Multithreading the SunOS Kernel”, *Proceedings of the 1992 Summer USENIX Conference*, San Antonio, TX, pp. 11-18, June 1992.

[19] H. Forbes and K. Schwan, “Rapid – A Multiprocessor Scheduler for Dynamic Real-Time Applications”, Technical Report GIT-CC-94-23, College of Computing, Georgia Institute of Technology, Apr. 1994.

[20] N. G. Fosback, *Stock Market Logic*, Institute for Econometric Research, Ft. Lauderdale, FL, 1976.

[21] M. R. Garey and D. S. Johnson, “Scheduling Tasks with Nonuniform Deadlines on Two Processors”, *JACM*, 23(3), pp. 461-467, July 1976.

[22] L. Georgiadis, R. Guérin, V. Peris, and K. N. Sivarajan, “Efficient Network QoS Provisioning Based on per Node Traffic Shaping”, *IEEE/ACM Transactions on Networking*, 4(4), pp. 482-501, Aug. 1996.

[23] D. B. Golub, “Operating System Support for Coexistence of Real-Time and Conventional Scheduling”, Technical Report CMU-CS-94-212, School of Computer Science, Carnegie Mellon University, Nov. 1994.

[24] P. Goyal, X. Guo, and H. M. Vin, “A Hierarchical CPU Scheduler for Multimedia Operating Systems”, *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, pp. 107-122, Oct. 1996.

[25] P. Goyal, Panel talk at the *IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems*, Washington, DC, Dec. 1996.

[26] J. G. Hanko, E. M. Kuerner, J. D. Northcutt, and G. A. Wall, "Workstation Support for Time-Critical Applications", *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Germany, pp. 4-9, Nov. 1991.

[27] J. G. Hanko, "A New Framework for Processor Scheduling in UNIX", Abstract talk at the *Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Lancaster, U. K., Nov. 1993.

[28] G. J. Henry, "The Fair Share Scheduler", AT&T Bell Laboratories Technical Journal, 63(8), pp. 1845-1858, Oct. 1984.

[29] *IEEE Micro*, 15(4), Aug. 1996.

[30] K. Jeffay and D. Bennett, "A Rate-Based Execution Abstraction for Multimedia Computing", *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Durham, NH, pp. 67-78, Apr. 1995.

[31] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera III, "Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System", *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Durham, NH, pp. 55-65, Apr. 1995.

[32] M. B. Jones, J. S. Barrera III, A. Forin, P. J. Leach, D. Rosu, and M. C. Rosu, "An Overview of the Rialto Real-Time Architecture", *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, pp. 249-256, Sept. 1996.

[33] M. B. Jones, personal communication, July 1997.

[34] M. B. Jones, D. Rosu, and M-C. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, pp. 198-211, Oct. 1997.

[35] K. B. Kenny and K. Lin, "Building Flexible Real-Time Systems Using the Flex Language", *IEEE Computer*, 24(5), pp. 70-78, May 1991.

[36] S. Khanna, M. Sebree, and J. Zolnowsky, "Realtime Scheduling in SunOS 5.0," *Proceedings of the 1992 Winter USENIX Conference*, San Francisco, CA, pp. 375-390, Jan. 1992.

[37] L. Kleinrock, *Queueing Systems Vol 2: Computer Applications*, John Wiley & Sons, Inc., New York, NY, 1976.

[38] C. J. Lindblad, "A Programming System for the Dynamic Manipulation of Temporally Sensitive Data", Technical Report MIT/LCS/TR-637, Laboratory for Computer Science, Massachusetts Institute of Technology, Aug. 1994.

[39] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.

[40] J. P. Lehoczky, L. Sha, and J. K. Strosnider, “Enhanced Aperiodic Responsiveness in Hard Real-Time Environments”, *Proceedings of the IEEE Real-Time Systems Symposium*, San Jose, CA, pp. 261-270, Dec. 1987.

[41] J. P. Lehoczky, L. Sha, and Y. Ding, “The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior”, *Proceedings of the IEEE Real-Time Systems Symposium*, Santa Monica, CA, pp. 166-171, Dec. 1989.

[42] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, “The Design and Implementation of an Operating System to Support Distributed Multimedia Applications”, *IEEE JSAC*, 14(7), pp. 1280-1297, Sept. 1996.

[43] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, “Policy/Mechanism Separation in Hydra”, *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 132-140, Nov. 1975.

[44] C. J. Lindblad, “A Programming System for the Dynamic Manipulation of Temporally Sensitive Data”, Technical Report MIT-LCS-TR-637, Laboratory for Computer Science, Massachusetts Institute of Technology, Aug. 1994.

[45] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, *JACM*, 20(1), pp. 46-61, Jan. 1973.

[46] C. D. Locke, “Best-Effort Decision Making for Real-Time Scheduling”, Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, May 1986.

[47] C. W. Mercer, S. Savage, and H. Tokuda, “Processor Capacity Reserves: Operating System Support for Multimedia Applications”, *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, pp. 90-99, May 1994.

[48] *Merriam-Webster’s Collegiate Dictionary*, 10th ed., Merriam-Webster, Springfield, MA, 1998.

[49] J. D. Northcutt, J. G. Hanko, A. T. Ruberg, and G. A. Wall, “NetCam: An Audio/Video Network-Attached Device”, In preparation, Dec. 1998.

[50] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, “SVR4 UNIX Scheduler Unacceptable for Multimedia Applications”, *Proceedings of the Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Lancaster, U. K., pp. 35-48, Nov. 1993.

[51] J. Nieh and M. S. Lam, “SMART UNIX SVR4 Support for Multimedia Applications”, *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Ottawa, Canada, pp. 404-414, June 1997.

[52] J. Nieh and M. S. Lam, “The Design, Implementation, and Evaluation of SMART: A Scheduler for Multimedia Applications”, *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, pp. 184-197, Oct. 1997.

[53] J. Nieh and M. S. Lam, “Multimedia on Multiprocessors: Where’s the OS When You Really Need It?”, *Proceedings of the Eighth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Cambridge, U. K., pp. 103-106, July 1998.

[54] J. D. Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*, Academic Press, Boston, MA, 1987.

[55] J. D. Northcutt, “The Alpha Operating System: Requirements and Rationale”, Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, Jan. 1988.

[56] J. D. Northcutt and E. M. Kuerner, “System Support for Time-Critical Applications”, *Proceedings of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Heidelberg, Germany, pp. 242-254, Nov. 1991.

[57] A. K. Parekh and R. G. Gallager, “A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case”, *IEEE/ACM Transactions on Networking*, pp. 344-357, June 1993.

[58] “PointCast Unveils First News Network that Reaches Viewers at Their Desktops”, Press Release, PointCast Inc., San Francisco, CA, Feb. 13, 1996.

[59] S. Ramos-Thuel and J. P. Lehoczky, “On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems”, *Proceedings of the IEEE Real-Time Systems Symposium*, Raleigh Durham, NC, pp. 160-171, Dec. 1993.

[60] R. W. Scheifler and J. Gettys, “The X Window System”, *ACM Transactions on Graphics*, 5(2), pp. 79-109, Apr. 1986.

[61] B. K. Schmidt, J.D. Northcutt, and M. S. Lam, “A Method and Apparatus for Measuring Media Synchronization”, *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video,* Durham, NH, pp. 203-214, Apr. 1995.

[62] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2nd ed., Addison-Wesley, Reading, MA, 1992.

[63] L. Sha, J. P. Lehoczky, and R. Rajkumar, “Solutions for Some Practical Problems in Prioritized Preemptive Scheduling”, *Proceedings of the IEEE Real-Time Systems Symposium*, New Orleans, LA, pp. 181-191, Dec. 1986.

[64] SLIC-Video User’s Guide, Rel. 1.0, MultiMedia Access Corporation, 1995.

[65] I. Stoica and H. Abdel-Wahab, “Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation”, Technical Report 95-22, Department of Computer Science, Old Dominion University, Nov. 1995.

[66] I. Stoica, H. Abdel-Wahab, and K. Jeffay, “On the Duality between Resource Reservation and Proportional Share Resource Allocation”, *Multimedia Computing and Networking Proceedings, SPIE Proceedings Series*, Vol. 3020, San Jose, CA, pp. 207-214, Feb. 1997.

[67] J. C. Tang and E. A. Isaacs, “Why Do Users Like Video? Studies of Multimedia-Supported Collaboration”, Technical Report SMLI TR-92-5, Sun Microsystems Laboratories, Dec. 1992.

[68] *UNIX System V Release 4 Internals Student Guide,* Vol. I, Unit 2.4.2., AT&T, 1990.

[69] C. A. Waldspurger, “Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management”, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.

[70] G. A. Wall, J. G. Hanko, and J. D. Northcutt, “Bus Bandwidth Management in a High Resolution Video Workstation,” *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video*, La Jolla, CA, pp. 274-288, Nov. 1992.

[71] T. Winograd, personal communication, Mar. 1993.

[72] J. Zolnowsky, “Realtime Dispatch in SunOS: an Update”, *Proceedings of the Second Annual SunSoft Technical Conference*, Menlo Park, CA, Apr. 1996.