A SMART Scheduler for Multimedia Applications

JASON NIEH
Columbia University
and
MONICA S. LAM
Stanford University

Real-time applications such as multimedia audio and video are increasingly populating the workstation desktop. To support the execution of these applications in conjunction with traditional non-real-time applications, we have created SMART, a Scheduler for Multimedia And Real-Time applications. SMART supports applications with time constraints, and provides dynamic feedback to applications to allow them to adapt to the current load. In addition, the support for real-time applications is integrated with the support for conventional computations. This allows the user to prioritize across real-time and conventional computations, and dictate how the processor is to be shared among applications of the same priority. As the system load changes, SMART adjusts the allocation of resources dynamically and seamlessly. It can dynamically shed real-time computations and regulate the execution rates of real-time tasks when the system is overloaded, while providing better value in underloaded conditions than previously proposed schemes.

We have implemented SMART in the Solaris UNIX operating system and measured its performance against other schedulers commonly used in research and practice in executing real-time, interactive, and batch applications. Our experimental results demonstrate SMART's superior performance over fair queueing and UNIX SVR4 schedulers in supporting multimedia applications.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Scheduling, real-time, multimedia, proportional sharing

Parts of this work appeared as a conference publication in the Sixteenth ACM Symposium on Operating Systems Principles [Nieh and Lam 1997b]. This work was supported in part by an NSF CAREER award and Sun Microsystems.
Authors' addresses: J. Nieh, Columbia University, 518 Computer Science Building, Mail Code 0401, 1214 Amsterdam Avenue, New York, NY 10027; email: nieh@cs.columbia.edu; M. S. Lam, Stanford University, Room 307, Gates Computer Science Building 3A, Stanford, CA 94305; email: lam@cs.stanford.edu.

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 [IEEE 1996].

While hardware has advanced to meet the special demands of multimedia applications, software environments have not. In particular, multimedia applications have real-time constraints which are not handled well by today's general-purpose operating systems. The problems experienced by users of multimedia on these machines include video jitter, poor "lip-synchronization" between audio and video, and slow interactive response while running video applications. Commercial operating systems such as UNIX SVR4 [AT&T 1990] attempt to address these problems by providing a real-time scheduler in addition to a standard time-sharing scheduler. However, such hybrid schemes lead to experimentally demonstrated unacceptable behavior, allowing runaway real-time activities to cause basic system services to lock up, and the user to lose control over the machine [Nieh et al. 1993].

This paper argues for the need to design a new processor scheduling algorithm that can handle the mix of applications we see today. We present a scheduling algorithm that we have implemented in the Solaris UNIX operating system [Eykholt et al. 1992], and demonstrate its improved performance over existing schedulers in research and practice on real applications. In particular, we have quantitatively compared it against the popular weighted fair queueing and UNIX SVR4 schedulers in supporting multimedia applications in a realistic workstation environment.

1.1 Demands of Multimedia Applications on Processor Scheduling

To understand the requirements imposed by multimedia applications on processor scheduling, we first describe the salient features of these applications and their special demands that distinguish them from the conventional (non-real-time) applications that current operating systems are designed for:

—*Soft real-time constraints*. Real-time applications have application-specific timing requirements that need to be met [Northcutt 1987]. For example in the case of video, time constraints arise due to the need to display video in a smooth and synchronized way, often synchronized with audio. Time constraints may be periodic or aperiodic in nature. Unlike conventional applications, tardy results are often of little value; it is often preferable to skip a computation than to execute it late. Unlike hard real-time environments, missing a deadline only diminishes the quality of the results and does not lead to catastrophic failures.

—*High resource demands and frequent overload*. Multimedia applications can present very high demands for resources. Today, video applications are often limited to simple VCR-like functions as opposed to delivering richer video processing functionality, and video playback windows are often tiny at full display rate because of computing resources insufficient to keep up with high-fidelity resolution. Since 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 often exceeding its availability.

—*Dynamically adaptive applications*. When resources are overloaded and not all time constraints can be met, multimedia applications are often able to adapt and degrade gracefully by offering a different quality of service [Northcutt and Kuerner 1991]. For example, a video application may choose to skip some frames or display at a lower image quality when not all frames can be processed in time. Because not all multimedia applications are written with adaptive capabilities, adaptive and non-adaptive multimedia applications must be able to co-exist.

—*Co-existence with conventional computations*. Real-time applications must share the desktop with already existing conventional applications, such as word processors, compilers, and so on. Real-time tasks should not always be allowed to run in preference to all other tasks because they may starve out important conventional activities, such as those required to keep the system running. Moreover, users would like to be able to combine real-time and conventional computations in new applications, such as multimedia documents, which mix text and graphics as well as audio and video. In no way should the capabilities of a multiprogrammed workstation be reduced to a single function commodity television set in order to meet the demands of multimedia applications.

—*Dynamic environment*. Unlike static embedded real-time environments, workstation users run an often changing mix of applications, resulting in dynamically varying loads.

—*User preferences*. Different users may have different preferences, for example, in regard to trading off the speed of a compilation versus the display quality of a video, depending 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.

1.2 Overview of this Paper

This paper proposes SMART (Scheduler for Multimedia And Real-Time applications), a processor scheduler that fully supports the application characteristics described above. SMART consists of a simple application interface and a scheduling algorithm that tries to deliver the best overall value to the user. SMART supports applications with time constraints, and provides dynamic feedback to applications to allow them to adapt to the current load. In addition, the support for real-time applications is integrated with the support for conventional computations. This allows the user to prioritize across real-time and conventional computations, and dictate how the processor is to be shared among applications of the same priority. As the system load changes, SMART adjusts the allocation of resources dynamically and seamlessly. It is able to dynamically shed real-time computations and regulate the execution rates of real-time tasks when the system is overloaded, while providing better value in underloaded conditions than previously proposed schemes.

SMART achieves this behavior by reducing this complex resource management problem into two decisions, one based on *importance* to determine the overall resource allocation for each task, and the other based on *urgency* to determine when each task is given its allocation. SMART provides a common importance attribute for both real-time and conventional tasks based on priorities and weighted fair queueing (WFQ) [Demers et al. 1989]. SMART then uses an urgency mechanism based on earliest-deadline scheduling [Liu and Layland 1973] to optimize the order in which tasks are serviced to allow real-time tasks to make the most efficient use of their resource allocations to meet their time constraints. In addition, a bias on conventional batch tasks that accounts for their ability to tolerate more varied service latencies is used to give interactive and real-time tasks better performance during periods of transient overload.

This paper also presents experimental data on the SMART algorithm, based on our implementation of the scheduler in the Solaris UNIX operating system. We present two sets of data, both of which are based on a workstation workload consisting of real multimedia applications running with representative batch and interactive applications. For the multimedia application, we use a synchronized media player developed by Sun Microsystems Laboratories that was originally tuned to run well with the UNIX SVR4 scheduler. It takes only the addition of a couple of system calls to allow the application to take advantage of SMART's features. We will describe how this is done to give readers a better understanding of the SMART application interface. The first experiment compares SMART with two other existing scheduling algorithms: UNIX SVR4 scheduling, which serves as the most common basis of workstation operating systems used in current practice [Ffoulkes and Wikler 1997], and WFQ, which has been the subject of much attention in current research [Bennett and Zhang 1996; Demers et al. 1989; Parekh and Gallager 1993; Stoica et al. 1996; Waldspurger 1995]. The experiment shows that SMART is superior to the other algorithms in the case of a workstation overloaded with real-time activities. In the experiment, SMART delivers over 250% more real-time multimedia data on time than UNIX SVR4 timesharing and over 60% more real-time multimedia data on time than WFQ, while also providing better interactive response. The second experiment demonstrates the ability of SMART to (1) provide the user with predictable control over resource allocation, (2) adapt to dynamic changes in the workload and (3) deliver expected behavior when the system is not overloaded.

The paper is organized as follows. Section 2 introduces the SMART application interface and usage model. Section 3 describes the SMART scheduling algorithm. We start with the overall rationale of the design and the major concepts, then present the algorithm itself, followed by an example to illustrate the algorithm. Despite the simplicity of the algorithm, the behavior it provides is rather rich. Section 4 analyzes the different aspects of it and shows how it delivers behavior consistent with its principles of operations. Section 5 provides a comparison with related work. Section 6 describes our implementation of SMART in the Solaris operating system. Section 7 presents experimental results based on using our SMART prototype implementation with real applications in a commercial operating system environment. Finally, we present some concluding remarks and directions for future work.

2. THE SMART INTERFACE AND USAGE MODEL

The SMART 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. For the application developer, SMART provides *time constraints* and *notifications* for supporting applications with real-time computations. For the user of applications, SMART provides *priorities* and *shares* for predictable control over the allocation of resources. An overview of the interface is presented here. A more detailed description can be found in Nieh and Lam [1997a].

2.1 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 them 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 two kinds of programming constructs for supporting applications with real-time computations: a time constraint to allow an application to express to the scheduler the timing requirements of a given block of application code, and a notification to allow the scheduler to inform the application via a simple upcall when its timing requirements cannot be met.

A time constraint consists of a deadline and an estimate of the processing time required to meet the deadline. An application can inform the scheduler that a given block of code has a certain deadline by which it should be completed, can request information on the availability of processing time for meeting a deadline, and can request a notification from the scheduler if it is not possible for the specified deadline to be met. Furthermore, 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 conventional computations. By providing explicit time constraints, SMART allows applications to communicate their timing requirements to the system. The scheduler can then use this information to 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.

A notification is used to allow an application to request that the scheduler inform it whenever its deadline cannot be met. A notification consists of a notify-time and a notify-handler. 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. 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 it, or perhaps change the time constraints. This feedback from the system enables adaptive real-time applications to degrade gracefully. 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.

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 and notifications 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. As an example, we describe an audio/video application that was programmed using time constraints in Section 7.4.

2.2 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 timesharing, all that a user is given is a `nice` knob [AT&T 1990] whose setting is poorly correlated to the scheduler's externally observable behavior [Nieh et al. 1993].

As users may have different preferences for how processing time should be allocated among a set of applications, 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, meaning that 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 has 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 at the same priority, the user can specify the share of each application, resulting in each application receiving an allocation of resources in proportion to its respective share whenever there is contention for resources. Shares only affect the allocation of resources among applications with equal priorities.

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 occasionally wish to adjust the proportion of shares between the applications. A 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 can always execute; this can be achieved by running the application with high priority.

2.3 Interface Design Summary

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 demands on either application developers or end users for information they cannot or choose not to 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. When the real-time programming support is desired, as is often the case with multimedia applications, SMART has the ability to provide it.

3. THE SMART SCHEDULER

In the following, we first describe the principles of operations used in the design of the scheduler. We then give an overview of the rationale behind the design, followed by an overview of the algorithm and then the details.

3.1 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 tasks 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, particularly by being able to effectively utilize available resources when the system is heavily loaded.

—*Satisfying real-time constraints and fast interactive response time in underload*. If real-time and interactive tasks 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 can cause many more deadlines to be missed and deliver poor response time to short running tasks. We will tolerate some instantaneous unfairness so long as the extent of the unfairness is bounded. For example, a long-running batch application can tolerate some extra short-term delay without any noticeable loss in overall performance to allow a real-time application to meet its immediate deadline. This is the same motivation behind the design of multi-level feedback schedulers [Leffler et al. 1989] to improve the response time of interactive tasks.

—*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.

3.2 Rationale and Overview

Real-time and conventional applications have very diverse characteristics. Real-time applications have some well-defined computation that must be completed before an associated deadline. The goal of real-time applications is simply to complete their computations before their respective deadlines. If it is not possible to meet all deadlines, it is generally better to complete as many computations as possible by their respective deadlines. For some real-time applications, it may be better not to run the application at all if it cannot meet all of its deadlines. 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 I. It is this diversity that makes devising an integrated scheduling

Table I. Categories of Applications

	Real-Time Applications	Conventional Applications	
		Interactive	Batch
Deadlines	Yes	No	No
Quantum of Execution	Service time: no value if the entire computation not executed	Arbitrary choice	Arbitrary choice
Resource Requirement	Service time before deadline; 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

algorithm difficult. A real-time scheduler uses real-time constraints to determine the execution order, but conventional tasks do not have real-time constraints. Adding periodic deadlines to conventional tasks is a tempting design choice, but it introduces artificial constraints that reduce the effectiveness of the system. On the other hand, a conventional task 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.

The crux of the solution is not to confuse urgency with importance. An urgent task is one that has an immediate real-time constraint. An important task is one with a high priority, or one that has been the least serviced proportionally among applications with the same priority. An urgent task may not be the one to execute if it requests more resources than its fair share. Conversely, an important task need not be run immediately. For example, a real-time task that has a higher priority but a later deadline may be able to tolerate the execution of a lower priority task 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 task 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 task has been allotted its proportional share of resources. The formal definition of the BVFT is given in Section 3.3. We say that task *A* has a higher value-tuple than task *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 task to run is as follows:

(1) If the task with the highest value-tuple is a conventional task (a task without a deadline), schedule that task.
(2) Otherwise, create a candidate set consisting of all real-time tasks with higher value-tuple than that of the highest value-tuple conventional task. (If no conventional tasks are present, all the real-time tasks are placed in the candidate set.)
(3) Apply the best-effort real-time scheduling algorithm [Locke 1986] 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 task with the earliest deadline whose execution does not cause any tasks with higher value-tuples to miss their deadlines. This is achieved by considering each candidate in turn, starting with the one having the highest value-tuple. The algorithm attempts to schedule the candidate into a working schedule that is initially empty. The candidate is inserted in deadline order in this schedule provided its execution does not cause any of the tasks in the schedule to miss its deadline. The scheduler simply picks the task with the earliest deadline in the working schedule.
(4) If a task 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.

3.3 Biased Virtual Finishing Time

The notion of a *virtual finishing time (VFT)*, which measures the degree to which the task has been allotted its proportional share of resources, has been previously used in describing fair queueing algorithms [Bennett and Zhang 1996; Demers et al. 1989; Parekh and Gallager 1993; Stoica et al. 1996; Waldspurger 1995]. 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 in the following ways. First, our use of virtual finishing times incorporates tasks 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 tasks 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—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 tasks into queues, one for each priority. The tasks in each queue are ordered in increasing BVFT values. Each task 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 task being executed has share S and was initiated at time τ. Let $v(\tau)$ denote the task's virtual time at time τ. Then the virtual time $v(t)$ of the task 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 tasks is executing. The rate of advance is proportional to the amount of processing time spent on the task divided by total number of shares of all tasks on the queue. To be more precise, suppose the current task being executed has priority P and was initiated at time τ. Let $V_P(\tau)$ denote the queue virtual time of the queue with priority P at time τ. 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 on the queue with priority P.

Previous work in the domain of packet switching provides a theoretical basis for using the difference between the virtual time of a task and the queue virtual time as a measure of whether the respective task has consumed its proportional allocation of resources [Demers et al. 1989; Parekh and Gallager 1993]. If a task'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 task 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 tasks on the queue, the relative magnitudes of the virtual times provide a measure of the degree to which each task 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 task is the unit of time the scheduler gives to the task to run before being rescheduled. The quantum for a real-time task 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 a task executes and uses up its time quantum, but only changes when the task 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 a task. Let the task in question have priority P and share S. Let $\beta(t)$ denote the BVFT of the task at time t.

Task creation time. When a task is created at time τ_0, it acquires as its virtual time the queue virtual time of its corresponding queue. Suppose the task has time quantum Q, then its BVFT is

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

Completing a Quantum. Once a task is created, its BVFT is updated as follows. When a task finishes executing for its time quantum, it is assigned a new time quantum Q. As a conventional task 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 t be the time a task's BVFT was last changed. In other words, if the bias has not changed since the time a task's BVFT was last changed, the increased bias b is zero. Then, the task's BVFT is

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

The bias is used to defer long running batch computations during transient loads to allow real-time and interactive tasks 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 [AT&T 1990]. 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 tasks 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 that is used in multilevel-feedback schedulers. However, while multilevel-feedback affects the actual average amount of resources allocated to each task, bias only affects the response time of a task 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 task 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 task a limited amount of credit for not using the processor cycles and to improve the responsiveness of these tasks. Therefore, SMART allows the task to remain on its given priority queue for a limited duration that is equal to the lesser of the deadline of the task (if one exists), or a system default. At the end of this duration, a sleeping task must leave the queue, and SMART records the difference between the task's and the queue's virtual time. This difference is then restored when the task rejoins the queue once it becomes runnable. Let E be the execution time the task has already received toward completing its time quantum Q, B be its current bias, and $v(t)$ denote the task's virtual time. Then, the difference δ 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 a task 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 task is given a new share, the BVFT is calculated by having the task 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.

3.4 Best-Effort Real-Time Scheduling

SMART iteratively selects tasks 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 task 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 task 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 task j to meet its deadline, and let E_j be the execution time task j has already spent running toward meeting its deadline. Let F_j be the fraction of the processor required by a periodic real-time task; F_j is simply the ratio of a task's service time to its period if it is a periodic real-time task, and zero otherwise. Let D_j be the deadline of the task. Then, the estimated resource requirement of task 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 task i in the schedule with deadline D_i, the following inequality holds:

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

On each task insertion into the working schedule, the resulting working schedule that includes the newly inserted task is tested for feasibility. If the resulting working schedule is feasible and the newly inserted task is a periodic real-time task, its estimate of future processing time requirements is accounted for in subsequent feasibility tests. At the same time, lower value-tuple tasks 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 tasks to miss their deadlines. The iterative selection process is repeated until SMART runs out of tasks or until it determines that no further tasks can be inserted into the schedule feasibly. Once the iterative selection process has been terminated, SMART then executes the earliest-deadline runnable task in the schedule. SMART further uses the selection process for determining which tasks cannot complete before their respective deadlines when the system is overloaded and notifies those tasks of the missed deadlines according to their notification parameters.

If there are no runnable conventional tasks and there are no runnable real-time tasks that can complete before their deadlines, the scheduler runs the highest value-tuple runnable real-time task, 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 task, even if it cannot satisfy its deadline.

3.5 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 tasks. 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 tasks of higher value than the highest value conventional task. This worst case occurs if each real-time task needs to be selected and feasibility tested against all other tasks when rebuilding the working schedule. It is unlikely for the worst case to occur in practice for any reasonably large N_R. Real-time tasks typically have short deadlines so that if there are a large number of real-time tasks, the scheduler will determine that there is no more slack in the schedule before all of the tasks need to be individually tested for insertion feasibility. The presence of conventional tasks in the workstation environment also prevents NR from growing large. For large

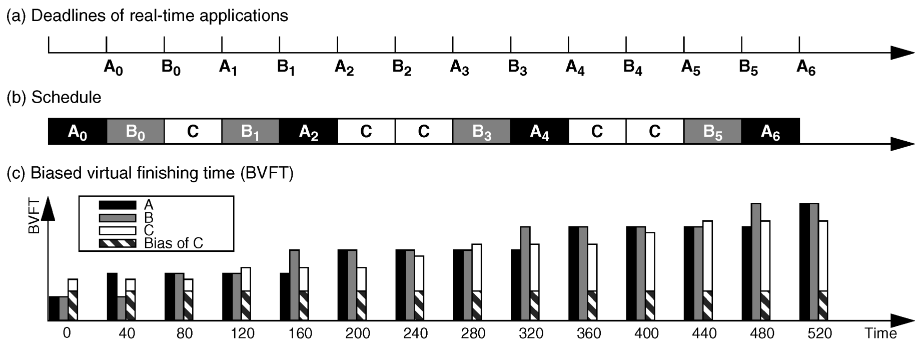

Fig. 1. Example illustrating the behavior of SMART.

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 tasks 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 $O(1)$.

3.6 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 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. For simplicity, we assume that the bias of C remains constant throughout the example. Figures 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.

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 tasks with the same BVFT; in general, a task with an early deadline may get to run over a task with an earlier BVFT but a later deadline.)

When a task 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 1(c) shows how the tasks are scheduled such that their BVFTs 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 task. 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_1 deadline, it executes application B instead and notifies A of the missed deadline.

4. 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 3.1. From the principles, priority is discussed in Section 4.1, proportional sharing and support for fluctuations in load are discussed in Section 4.2, satisfying real-time constraints is discussed in Section 4.2.2, trading off instantaneous fairness for better response time is discussed in Section 4.2.1, and notification of resource availability is discussed in Section 4.2.2.

4.1 Priority

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

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 task with an earlier deadline first, only if it can determine that doing so does not cause the high priority task 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.

4.2 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 tasks, we simply divide the cycles in proportion to the shares across the different applications. As noted in Table I, 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.

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

A bias is added to a task'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.

4.2.2 *Real-Time Applications.* We say that a system is *underloaded* if there are sufficient cycles to give a fair share to the conventional tasks in the system while satisfying all the real-time constraints. When a system is underloaded, the conventional tasks 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 tasks 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 [Liu and Layland 1973], which has been shown to satisfy all scheduling constraints, periodic or aperiodic, optimally [Dertouzos 1974]. A real-time scheduler is considered optimal if it can meet the deadlines of all tasks whenever such a schedule exists. In particular in this paper, we refer to the optimality of a scheduler with respect to real-time tasks only, assuming no conventional tasks are present.

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 tasks, it scales down the real-time tasks proportionally. In addition, the bias introduced in the algorithm, as well as the use of a best-effort scheduler among real-time tasks with sufficiently high values, allows more real-time constraints to be met.

5. RELATED WORK

Recognizing the need to provide better scheduling to support the needs of modern applications such as multimedia, a number of resource management mechanisms have been proposed. These approaches can be loosely classified as real-time scheduling, fair queueing, feedback-based allocation, and hierarchical scheduling. We discuss these approaches in the context of supporting multimedia applications.

5.1 Real-Time Scheduling

Real-time schedulers such as rate-monotonic scheduling [Lehoczky et al. 1989; Liu and Layland 1973] and earliest-deadline scheduling [Dertouzos 1974; Liu and Layland 1973] are designed to make better use of hardware resources in meeting real-time requirements. In particular, earliest-deadline scheduling is optimal in underload. These approaches are widely used in supporting real-time embedded systems. However, they do not perform well when the system is overloaded, nor are they designed to support conventional applications, which have limited their utility in more general-purpose computing environments.

Resource reservations are commonly combined with real-time scheduling in an attempt to run real-time tasks with conventional tasks [Coulson et al. 1995; Jones et al. 1997; Leslie et al. 1996; Mercer et al. 1994]. These approaches are used with admission control to allow real-time tasks to reserve a fixed percentage of the resource in accordance with their resource requirement. Any leftover processing time is allocated to conventional tasks using a standard timesharing or round-robin scheduler.

Several differences in these reservation approaches are apparent. While the approaches in Coulson et al. [1995] and Leslie et al. [1996] take advantage of earliest-deadline scheduling to provide optimal real-time performance in underload, the rate monotonic utilization bound used in Mercer et al. [1994] and the time interval assignment used in Rialto [Jones et al. 1997] are not optimal, resulting in lower performance than earliest-deadline approaches. Unless conventional tasks are also assigned reservations, starvation can be a problem. This problem is exacerbated in Rialto in which even in the absence of reservations, applications with time constraints buried in their source code are given priority over conventional applications [M. B. Jones, Personal communication].

The use of reservations relies on admission control and resource negotiation policies to avoid overload. Rather than shedding load dynamically during program execution, as SMART does, such systems typically allow applications to reserve their needed amounts of resources in advance in an attempt to ensure up front that all deadlines can be met. Like in SMART, if all resource needs cannot be met in a reservation system, load will have to be shed. The difference is that reservation systems attempt to shed the load up front at reservation time, rather than dynamically, during program execution. Reservation systems are thus better suited when it is possible to determine up front what resources are necessary and for applications needing to know up front that they should be able to meet all their deadlines. However, these systems are not designed to effectively support applications that can dynamically adapt to varying amounts of offered resources.

Most resource reservation schemes have used a first-come-first-served approach, which is simple but has obvious drawbacks. Such schemes result in later arriving applications being denied resources even if they are more important. Others have proposed allowing users to specify the relative importance of applications, much like SMART does, while acting on these user preferences at reservation time, rather than dynamically during program execution. However, we are not aware of any importance-based resource reservation scheduler implementations.

Unlike first-come first-served reservation approaches, best-effort real-time scheduling [Locke 1986] provides optimal performance in underload while ensuring that tasks of higher priority can meet their deadlines in overload. However, it provides no way of scheduling conventional tasks 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 [Baiceanu et al. 1996; Goyal et al. 1996].

5.2 Fair Queueing

Fair queueing provides a mechanism that allocates resources to tasks in proportion to their shares. It was first proposed for network packet scheduling in Demers et al. [1989], with a more extensive analysis provided in Parekh and Gallager [1993], and later applied to processor scheduling in Waldspurger [1995] as stride scheduling. Recent variants [Bennett and Zhang 1996; Stoica et al. 1996] provide more accurate proportional sharing at the expense of additional scheduling overhead. Fair queueing approaches have been particularly useful in the context of network routers in managing resources in the face of many competing flows.

Fair queueing can be effective at meeting real-time requirements if the resource requirements of the tasks are less than their respective assigned shares. However, determining a set of share assignments that achieves this for a set of tasks with dynamically changing resource requirements is difficult. This problem can be simplified if tasks are assumed to have strictly periodic resource requirements [Parekh and Gallager 1993; Stoica et al. 1997], but common multimedia applications such as the JPEG video player application discussed in Section 7.4 do not satisfy this property. Another approach would be to overprovision the share assignments for real-time tasks while sacrificing utilization.

Instead of assigning shares based on task resource requirements, shares can be assigned in accordance with user desired allocations [Waldspurger 1995]. For instance, all tasks can be given equal shares by default to provide fair default resource allocations. However in this case, fair queueing does not perform well for real-time tasks because it does not account for their time constraints. In underload, time constraints are unnecessarily missed when a fair share allocation is less than the resource requirements of a real-time task. In overload, all tasks are proportionally late, potentially missing all time constraints.

Unlike real-time reservation schedulers, fair queueing can integrate reservation support for real-time tasks with proportional sharing for conventional tasks [Stoica et al. 1997]. 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 can 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. Furthermore, SMART also supports simultaneous prioritized and proportional resource allocation. SMART subsumes fair queueing in that it provides fair queueing behavior when time constraints are not used and all tasks are at the same priority.

The notion of a bias used in SMART is similar to the delta used in conjunction with the virtual time in Demers et al. [1989]. SMART's bias is adjusted by the system to systematically improve the performance of interactive applications. More recent work [Duda and Cheriton 1999] has built on this idea by exploring the use of direct user-level controls for adjusting the bias in the absence of support for real-time applications.

5.3 Feedback-Based Allocation

More recently, feedback-based allocation [Steere et al. 1999] has been developed in conjunction with reservation-based scheduling. It monitors the progress of applications and uses that information to guide the allocation of resources. The contributions of this work are primarily in deriving the appropriate assignment of scheduling parameters for different applications based on their resource requirements, especially in the presence of application dependencies. For instance, if a producer application is frequently being prevented from making progress because the corresponding consumer application is not able to run frequently enough, the proposed system controller will provide feedback to the scheduler to increase the allocation of the consumer application. The system feedback controller provides an alternative framework for varying the allocation of resources to different applications yet avoiding the priority inversion problem [Lampson and Redell 1980] that arises with priority-based schemes. The primary contribution of this work is complementary to our work on scheduling mechanisms for allowing real-time and conventional applications to co-exist while efficiently using system resources for meeting real-time requirements.

5.4 Hierarchical Scheduling

Because creating a single scheduler to service both real-time and conventional resource requirements has proven difficult, a number of hybrid schemes [Bollella and Jeffay 1995; Custer 1993; Golub 1994; Goyal et al. 1996; AT&T 1990] have been proposed. These approaches statically separate scheduling policies for real-time and conventional applications, respectively. Hierarchical scheduling is supported in a number of commercial operating systems, including Windows, Solaris, and Linux, and has been useful for providing support for different scheduling policies in a single underlying scheduling framework. The policies are combined using either priorities [Custer 1993; Golub 1994; AT&T 1990] or proportional sharing [Bollella and Jeffay 1995; Goyal et al. 1996; Hanko 1993] as the base level scheduling mechanism. However, the method used for combining different policies can be a limitation with these approaches.

With priorities, all tasks scheduled by the real-time scheduling policy are assigned higher priority than tasks scheduled by the conventional scheduling policy. This causes all real-time tasks, regardless of whether or not they are important, to be run ahead of any conventional task. 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 [Nieh et al. 1993].

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 timeslices between them. Proportional-share hierarchical scheduling can provide the benefit of performance isolation among different scheduling policies, which can be very desirable in support of a system with several competing entities. However, while real-time applications will not take over the machine, they also cannot meet their time constraints as 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 trying to ensure fairness [Goyal 1996].

In the context of supporting real-time multimedia applications, 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 are limited in being able to propagate these decisions to the lowest-level of resource management where the actual scheduling of processor cycles takes place. This simplifies the scheduling framework and works well in providing proportional fairness or strict prioritization of scheduling policies. However, it can be a limitation in maximizing the overall system utility.

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 tasks 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.

6. IMPLEMENTATION

We have implemented SMART in Sun Microsystems's Solaris UNIX operating system, version 2.5.1. Our implementation provides interfaces between SMART and existing operating system code in a way that is backwards compatible with the existing Solaris scheduling framework. In this section, we describe some important implementation issues. Section 6.1 provides some necessary background about the two-level Solaris scheduling framework. Section 6.2 describes our implementation methodology, which was to replace the existing Solaris dispatcher and introduce a new scheduling class.

6.1 Solaris Scheduling Framework

The Solaris operating system is a multithreaded UNIX SVR4 conformant operating system. Unlike older UNIX systems that only provide kernel support for UNIX processes, Solaris scheduling is based on threads. 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 is 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 its execution history, 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 is 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.

6.2 SMART Scheduling Framework in Solaris

To implement SMART in the Solaris operating system, we replaced the existing priority dispatcher with a SMART dispatcher that incorporates information about a thread's deadlines and shares as well as priorities. We then created a new scheduling class to provide access for users and applications to the new functionality offered by SMART. Threads were used as the basic schedulable entity in our SMART framework to provide a natural mapping to the structure of the Solaris operating system.

6.2.1 *SMART Dispatcher.* 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 Solaris 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 place 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 discuss later how 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 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. The high-resolution timeout implementation is also hardware dependent. When the hardware provides a single cycle counter register that is automatically decremented to zero and reset to its initial value in hardware, the implementation simply reduces the initial value to incur a periodic 1 ms timer interrupt instead of a periodic 10 ms timer interrupt. When the hardware provides two registers, a cycle counter register that is incremented by hardware at each processor cycle and a compare register that is set by the operating system to trigger a timer interrupt when the register values are equal, the implementation sets the compare register to correspond to the next time-based event that requires a timer interrupt.

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.

6.2.2 *Legacy Scheduling Classes.* In addition to providing SMART functionality, the SMART dispatcher is designed to support legacy scheduling classes without modification of those classes. 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 function 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.

6.2.3 *SMART Scheduling Class.* 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 that is by default allowed to use the same range of priorities as the SVR4 TS 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, 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. Because notifications are implemented as signals for simplicity in the prototype, the delivery of notifications in the prototype follows the same semantics of signals as used in Solaris.

6.2.4 *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.

7. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the SMART scheduler, we conducted a number of experiments on our SMART prototype, running microbenchmarks as well as real 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. We describe the experimental testbed that was used for our experiments. We highlight some experiments with various microbenchmarks that demonstrate the range of behavior possible with the scheduler in a real system environment. We then focus on comparing the performance of SMART against two 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.

7.1 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. The high resolution timing functionality discussed in Section 6.2.1 was used for all of the schedulers 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 ms resolution. We measured the cost of the mechanism on the testbed workstation to be 2-4 ms 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.

Table II. Microbenchmark Experiments

Experiment	Duration	Tasks	Shares	R service time	R deadline	C bias
Conventional	1000 s	$C_1 : C_2 : C_3$	3 : 2 : 1	N/A	N/A	0
Real-time	80 s	$R_1 : R_2 : R_3$	3 : 2 : 1	20 ms	40 ms	0
Mix	80 s	$R_4 : C_1$	1 : 1	10-30 ms	40 ms	0-100 ms

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.

7.2 Microbenchmarks

We highlight some microbenchmark performance results for three mixes of real-time and conventional resource requests that illustrate SMART's behavior under a dynamically changing load. 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. In particular, we focus on the more novel 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 that the scheduler drops the minimum number of deadline requests to achieve fair sharing, in the case of overload. Finally, we also show that bias helps minimize the number of deadlines dropped.

We used a simple conventional compute-oriented application C and a simple real-time application R for our microbenchmark experiments. The conventional application C does some simple CPU computations for a configurable number of loop iterations without blocking. The real-time application R also does some simple CPU computations for a configurable number of loop iterations, but each loop iteration must be completed within a given time constraint. If the loop cannot complete before its deadline, it skips the given iteration. If the loop completes before its deadline, the application sleeps until the start of the next time constraint. The number of loop iterations, the periodic deadline of each time constraint, and the duration of the loop computation are all configurable. We configured these applications in three simple experiments listed in Table II.

In the first experiment listed in Table II, we ran C_1, C_2, and C_3, three instances of conventional application C with relative shares of the ratio 3 : 2 : 1. The applications were started at approximately the same time and each was configured with a running time of about 338 s. No bias was used for the conventional applications in this experiment.

If the system were perfect, we would expect the applications to obtain processing time in proportion to their shares and C_1 to complete first since it has the largest share. Since it is assigned half of the total shares, C_1 should obtain

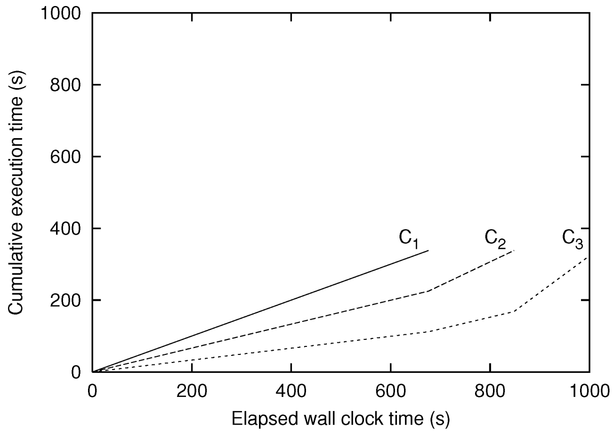

Fig. 2. Cumulative execution time of conventional applications.

half of the total processing time and complete in about 676 s. After C_1 completes, C_2 will obtain about two-thirds of the processing time since its share is twice as large as C_3. We would then expect C_2 to complete at about 846 s into the experiment. After C_2 completes, C_3 will obtain all of the processing time and should complete at about 1014 s into the experiment.

As shown in Figure 2, our measurements from the first experiment show that SMART delivers behavior close to what would be expected if the system were perfect. C_1, C_2, and C_3 complete their execution at times 678 s, 848 s, and 1018 s, respectively. Furthermore, the resource consumption rates of the applications were as expected throughout the experiment, 3 : 2 : 1 during the first 678 s of the experiment when all of the applications were running, and 2 : 1 during the next 170 s of the experiment when just C_2 and C_3 were running.

In the second experiment listed in Table II, we ran R_1, R_2, and R_3, three instances of real-time application R with relative shares of the ratio 3 : 2 : 1. Each real-time resource request took approximately 20 ms of execution time to complete, and each resource request had 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. R_1 processed a sequence of 1000 real-time requests, R_2 processed a sequence of 1500 real-time requests, and R_3 processed a sequence of 2000 real-time requests during the 80 s experiment. As a result, the applications completed their requests at different times during the experiment.

If the system were perfect, we would expect the three applications to accumulate processing time in accordance with their shares during the first 40 s of the experiment while the system is overloaded. In particular, we would expect R_1 to accumulate 20 s of processing time, enough to meet all of its deadlines until it completes. Because the system is overloaded, R_2 and R_3 would not be able to meet all of their deadlines but instead would meet deadlines in proportion to their shares, 66% and 33% of deadlines, respectively. During the second 40 s of the experiment after R_1 completes, we would expect R_2 and R_3 to be able to

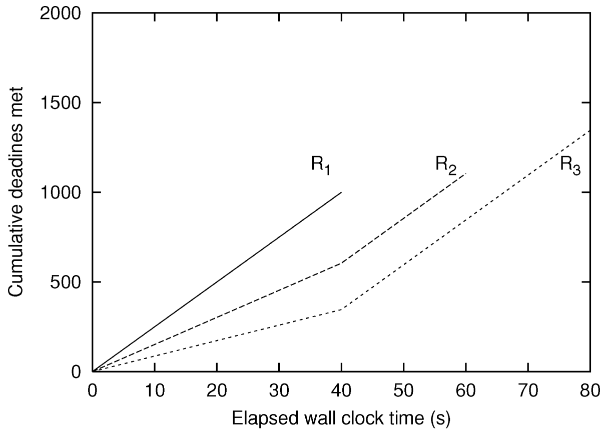

Fig. 3. Deadlines met by real-time applications.

meet the remainder of their deadlines since the system is no longer overloaded. Although R_2's share is twice as large as R_3's, the two applications will share processing time equally during the second 40 s of the experiment since R_2 does not need twice as much processing time as R_3.

As shown in Figure 3, our measurements from the second experiment show that SMART delivers behavior close to what would be expected if the system were perfect. During the first 40 s of the experiment, R_1, R_2, and R_3 consumed resources in proportion to their shares and were able to meet 99.9%, 60%, and 34% of their respective deadlines. The deviation from perfect behavior is in part caused by higher priority system threads that occasionally execute as part of the Solaris operating system. During the second 40 s of the experiment, both R_2 and R_3 missed a couple of deadlines as R_1 completed its execution, but then were able to meet all of their remaining deadlines. The results of this experiment illustrate SMART's behavior on real-time applications when the system is overloaded, when the system is underloaded, and when the load on the system is dynamically changing.

In the third experiment listed in Table II, we ran R_4, an instance of real-time application R, and C_1, an instance of conventional application C. with relative shares of the ratio 1 : 1. For R_4, each real-time resource request took on average 20 ms of execution time to complete, but the execution time varied between 10 to 30 ms. The deadline of each resource request was 40 ms from its instantiation, resulting in 2000 real-time resource requests over the 80 s experiment. In this experiment, we adjusted the SMART scheduler to allow us to fix the bias assigned to a conventional application. For C_1, we ran it with a bias of 0 ms and then repeated the experiment with a bias of 100 ms. Both applications were assigned equal shares.

Our measurements from the third experiment show that the bias can result in substantial improvement in performance for R_4 without changing the overall average resource allocation. When C_1 is given a bias of 0 ms in this experiment, R_4 misses 192 out of 2000 deadlines. However, when C_1 is given a bias of just 100 ms, R_4 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. On average, both R_4 and C_1 received 50% of the machine in our experiments regardless of the bias. Note that if the system were perfect and R_4 required 20 ms to process each 40 ms deadline request, we would expect R_4 to obtain 50% of the CPU and not miss any of its deadlines. We also ran this experiment and verified that SMART provided this performance.

7.3 Multimedia 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. 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 of them.

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 [Scheifler and Gettys 1986] 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. It adapts to its system environment by adjusting the quality of playback based on the system load. 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 are presented in Section 7.4. 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. The audio input stream contains standard 8-bit μ-law (CCIT standard G.711) monaural samples, as used in the AU audio file format popularized by Sun Microsystems. The captured data is from a satellite news network.

—*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.

7.4 Programming with Time Constraints

The SMART application interface makes it easier to develop a real-time application. The software developer can express the scheduling constraints directly to the system and have the system deliver the expected behavior. To illustrate this aspect of SMART, we first describe what it took to develop the IMS Player for UNIX SVR4, then discuss how we modified it for SMART.

7.4.1 *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, as in the case of *Entertain*, 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, as in the case of *News*, the video player attempts to synchronize its output with that of the audio device. 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, then it delays 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 [Fosback 1976] 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.

Figure 4 indicates 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. Note that MPEG video would require averaging for each type of frame. Each graph shows the actual execution time for each frame and the estimate error for each frame. The estimate error is the difference between the estimated and actual execution time for each frame. The slight positive bias in the difference is due to the 10% scaling in the estimate versus the actual execution time. As shown in the figure, there is a wide variance in the time it takes to handle a frame. The results also illustrate the difficulty of using a resource reservation scheme when resource requirements can vary substantially over time, as has been further demonstrated in Duda and Cheriton [1999]. Using the upper bound on the processing time as an estimate may yield

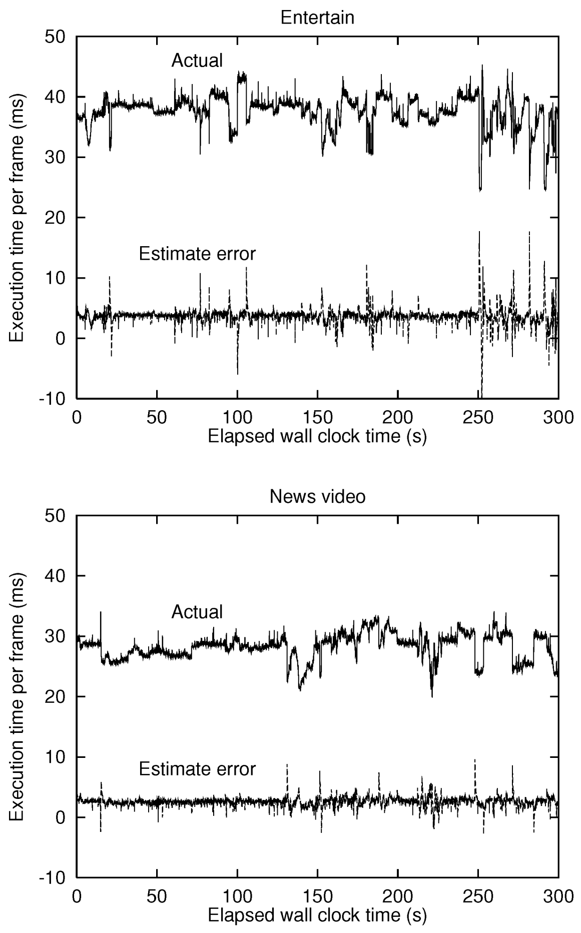

Fig. 4. Actual vs. estimated execution time per JPEG image.

a low utilization of resources; using the average processing time may cause too many deadlines to be missed.

7.4.2 *Audio Player.* The audio player reads a timestamped 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. Audio segments that cannot be processed before their deadlines are simply displayed late. Note that because of

Table III. Standalone Execution Times of Applications

Application	Measurement		CPU Time		Avg CPU
Name	Basis	Number	Avg	Std Dev	Utilization
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%

the workahead feature and the audio device buffering, the resulting deadlines can be highly aperiodic.

7.5 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 tasks, the throughput of batch tasks and the number of deadlines met in real-time tasks. 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 III 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.

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 tasks such as typing, cursor motion, or mouse selection, system response time should be less than 50–150 ms [Shneiderman 1992]. 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

Table IV. Standalone Application Quality Metric Performance

Name	Quality Metric	On Time	Early	Late	Dropped	Avg Time	Std Dev
News audio	Number of audio dropouts	100.00%	0.00%	0.00%	0.00%	0	0
News video	Actual minus desired display time	99.75%	0.09%	0.13%	0.02%	1.50 ms	2.54 ms
Entertain	Actual minus desired display time	99.58%	0.22%	0.13%	0.07%	1.95 ms	3.61 ms
Typing	Delay from character input to 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

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 IV shows the performance of each application when it was executed on its own.

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 [Schmidt 1995]. 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.

7.6 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 in this paper. 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 because there was 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 task it needs to execute. The scheduler simply selects the highest priority task to execute, with all tasks 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 tasks. 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 tasks 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 a task is 20 μs while the cost of reading the scheduling information for a task 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.

7.7 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:

—SVR4-RT: The real-time *News* and *Entertain* applications are put in the real-time class, leaving *Typing* and *Dhrystone* in the time-sharing class.
—SVR4-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 SVR4-TS, WFQ, and SMART, we expect the remainder of the computation time to be distributed evenly between *News* video, *Entertain*, and *Dhrystone*.

To provide a baseline for comparing scheduler performance, we define an ideal scheduler in this context as as one that makes the best use of each application's allocation toward maximizing that application's quality metric. For the default scheduler behavior in which applications are weighted equally, the ideal scheduler will be one that is fair and can perfectly use the allocated CPU cycles for the real-time applications toward meeting their deadlines. Because the system is overloaded, we expect that even for an ideal scheduler, the percentages of the frames dropped to be 25% and 45% for *News* video and *Entertain*, respectively.

Figure 5 presents the CPU allocation across different applications by different schedulers. It includes the percentage of the CPU used for executing other

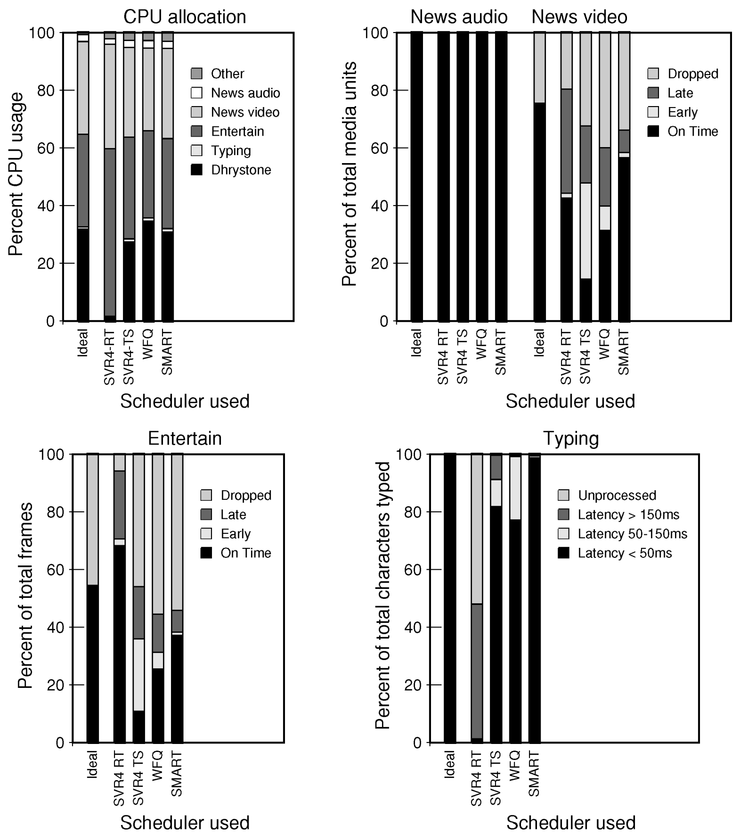

Fig. 5. Comparison of scheduler application performance.

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, the figure also shows the percentages of media units that are displayed on-time, early, late, or dropped. For the interactive *Typing* application, the figure 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. Figure 6 presents 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 figure. We observe that every scheduler handles the *News* audio application well with no audio dropouts. Because the audio application required little resources relative to the resource allocation given to it by each scheduler, all of the schedulers were able to meet its real-time constraints. In particular, the audio application performance illustrates SMART's ability to also meet real-time constraints by simply using a much larger share allocation than needed. Since there was no difference in the audio application performance among the schedulers, we will only concentrate on discussing the quality of the rest of the applications.

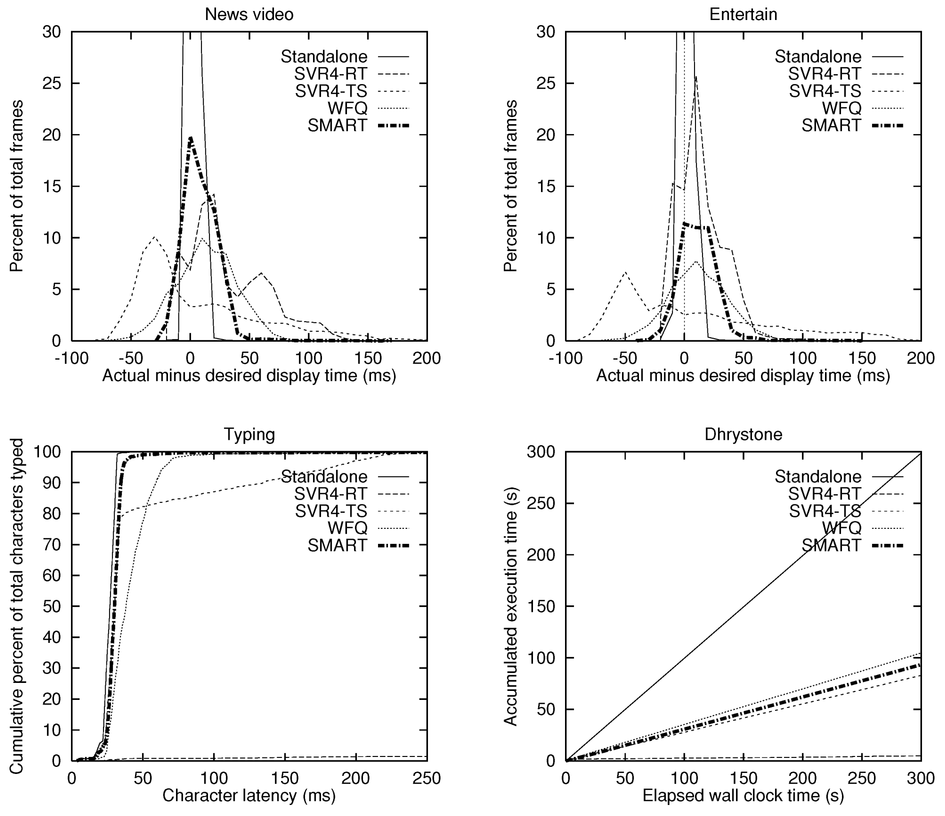

Fig. 6. Distributions of quality metrics.

Unlike the other schedulers, the SVR4-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 SVR4-RT for the *News* video, while using less resources.) Unfortunately, SVR4-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 tasks. Clearly, it is not acceptable to use the SVR4-RT scheduler.

All the other schedulers spread the resources relatively evenly across the three demanding applications. The SVR4-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 the 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 a task is based only on the time that the task is runnable and does not include any time that the task 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 among the schedulers in the number of frames dropped. 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.

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 SVR4-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 Figure 6, 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 delay 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.

In contrast, 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 SVR4-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 SVR4-TS are extremely late, causing many frames to be processed extremely early, resulting in a very large variance in display time across frames.

Finally, as shown in Figure 6, SMART is superior to both SVR4-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 a task does not accumulate any credit at all when it sleeps. We performed an experiment where the WFQ algorithm is modified to allow the blocked task 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.

7.8 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, 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.

Figure 7 shows the CPU allocation and quality metrics of the different applications run under SMART as well as an ideal scheduler. The figure shows 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

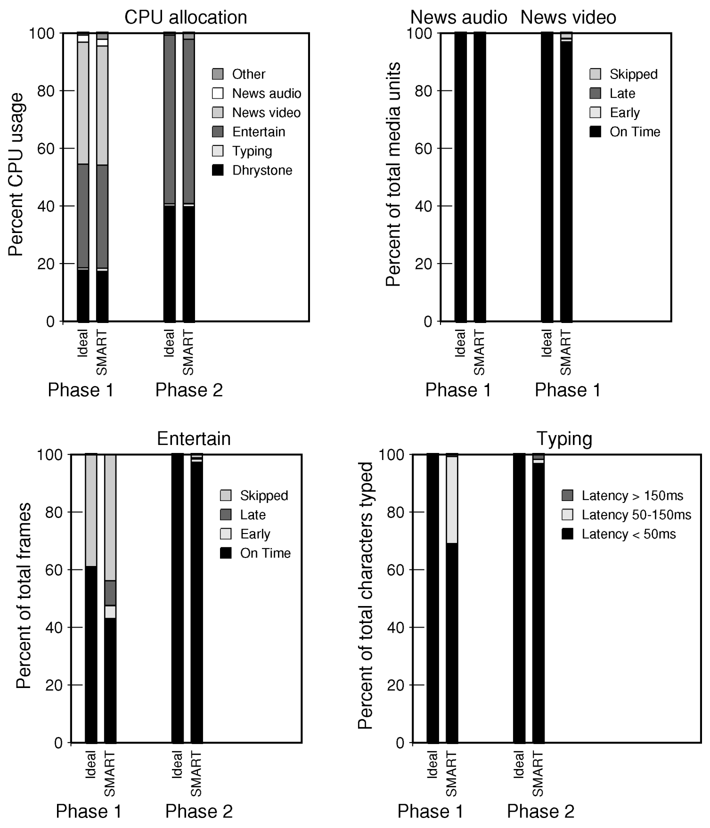

Fig. 7. SMART application performance under a changing load when using end user controls.

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 [Card et al. 1983].

8. CONCLUSIONS AND FUTURE WORK

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 widely-used UNIX SVR4 and fair queueing schedulers without requiring 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 weighted fair queueing and UNIX SVR4 schedulers in supporting multimedia applications in a realistic workstation environment.

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.

ACKNOWLEDGMENTS

We thank Jim Hanko, Duane Northcutt, and Brian Schmidt for their help with the applications and measurement tools used in our experiments. We also thank the anonymous reviewers for comments on earlier drafts that have helped to improve the paper.

REFERENCES

AT&T. 1990. *UNIX System V Release 4 Internals Student Guide*.

BAICEANU, V., COWAN, C., MCNAMEE, D., PU, C., AND WALPOLE, J. 1996. Multimedia Applications Require Adaptive CPU Scheduling. In *Proceedings of the IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems*. Washington, DC.

BENNETT, J. C. R. AND ZHANG, H. 1996. WF2Q: Worst-case Fair Weighted Fair Queueing. In *IEEE INFOCOM '96*. San Francisco, CA, 120–128.

BOLLELLA, G. AND JEFFAY, K. 1995. Support for Real-Time Computing Within General Purpose Operating Systems: Supporting Co-Resident Operating Systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. Chicago, IL, 4–14.

CARD, S. K., MORAN, T. P., AND NEWELL, A. 1983. *Psychology of Human-Computer Interaction*. L. Erlbaum Associates, Hillsdale, NJ.

COULSON, G., CAMPBELL, A., ROBIN, P., BLAIR, G., PAPATHOMAS, M., AND HUTCHINSON, D. 1995. The Design of a QoS Controlled ATM Based Communications System in Chorus. *IEEE J. Selected Areas Comm. (JSAC) 13*, 4 (May), 686–699.

CUSTER, H. 1993. *Inside Windows NT*. Microsoft Press, Redmond, WA.

DEMERS, A., KESHAV, S., AND SHENKER, S. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of SIGCOMM '89*. 1–12.

DERTOUZOS, M. 1974. Control Robotics: The Procedural Control of Physical Processors. In *Proceedings of the IFIP Congress*. Stockholm, Sweden, 807–813.

DUDA, K. AND CHERITON, D. 1999. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. Kiawah Island Resort, SC, 261–276.

EYKHOLT, J. R., KLEIMAN, S. R., BARTON, S., FAULKNER, R., SHIVALINGIAH, A., SMITH, M., STEIN, D., VOLL, J., WEEKS, M., AND WILLIAMS, D. 1992. Beyond Multiprocessing... Multithreading the SunOS Kernel. In *Proceedings of the 1992 Summer USENIX Conference*. San Antonio, TX, 11–18.

FFOULKES, P. AND WIKLER, D. 1997. Workstations Worldwide Market Segmentation. In *Advanced Desktops and Workstations Worldwide. Dataquest.*

FOSBACK, N. G. 1976. *Stock Market Logic*. Institute for Econometric Research, Ft. Lauderdale, FL.

GOLUB, D. B. 1994. Operating System Support for Coexistence of Real-Time and Conventional Scheduling. Tech. Rep. CMU-CS-94-212, School of Computer Science, Carnegie Mellon University. Nov.

GOYAL, P. 1996. Panel talk. In *IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems*. Washington, DC.

GOYAL, P., GUO, X., AND VIN, H. M. 1996. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*. Seattle, WA, 107–122.

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

IEEE. 1996. *IEEE Micro 15*, 4 (Aug.).

JONES, M. B., ROSU, D., AND ROSU, M.-C. 1997. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. St. Malo, France, 198–211.

LAMPSON, B. W. AND REDELL, D. D. 1980. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (Feb.), 105–117.

LEFFLER, S. J., MCKUSICK, M. K., KARELS, M. J., AND QUARTERMAN, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA.

LEHOCZKY, J., SHA, L., AND DING, Y. 1989. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*. 166–171.

LESLIE, I. M., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. 1996. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE J. Selected Areas Comm. (JSAC) 14*, 7 (Sept.), 1280–1297.

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM 20*, 1 (Jan.), 46–61.

LOCKE, C. D. 1986. Best-Effort Decision Making for Real-Time Scheduling. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University.

MERCER, C. W., SAVAGE, S., AND TOKUDA, H. 1994. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. Boston, MA, 90–99.

NIEH, J., HANKO, J. G., NORTHCUTT, J. D., AND WALL, G. A. 1993. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*. Lancaster, U. K., 35–48.

NIEH, J. AND LAM, M. S. 1997a. SMART UNIX SVR4 Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. Ottawa, Canada, 404–414.

NIEH, J. AND LAM, M. S. 1997b. The Design, Implementation and Evaluation of SMART: A scheduler for Multimedia Applications. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. ACM, St. Malo, France, 184–197.

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

NORTHCUTT, J. D. AND KUERNER, E. M. 1991. System Support for Time-Critical Applications. In *Proceedings of the Second International Workshop on Network and Operating Systems Support* *for Digital Audio and Video, Lecture Notes in Computer Science*. Vol. 614. Heidelberg, Germany, 242–254.

Parekh, A. K. and Gallager, R. G. 1993. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *IEEE/ACM Trans. Netw. 1*, 3 (June), 344–357.

Scheifler, R. W. and Gettys, J. 1986. The X Window System. *ACM Trans. Graph. 5*, 2 (Apr.), 79–109.

Schmidt, B. K. 1995. A Method and Apparatus for Measuring Media Synchronization. In *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video*. Durham, NH, 203–214.

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

Steere, D. C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Walpole, J. 1999. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. New Orleans, LA, 145–158.

Stoica, I., Abdel-Wahab, H., and Jeffay, K. 1997. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Multimedia Computing and Networking Proceedings, SPIE Proceedings Series*. Vol. 3020. San Jose, CA, 207–214.

Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S. K., Gehrke, J. E., and Plaxton, C. G. 1996. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the Seventeenth IEEE Real-Time Systems Symposium*. Washingtion, DC, 288–299.

Waldspurger, C. A. 1995. Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Received May 2001; revised March 2002, August 2002; accepted October 2002