Appears in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, October, 1997.

The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications

Jason Nieh[1,2] and Monica S. Lam[1]
[1]Computer Systems Laboratory, Stanford University
[2]Sun Microsystems Laboratories

Abstract

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. SMART is unique in its ability to automatically shed real-time tasks and regulate their execution rates 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 in executing real-time, interactive, and batch applications. Our results demonstrate SMART's superior performance in supporting multimedia applications.

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 [20].

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 [39] 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 [29].

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 which we have implemented in the Solaris UNIX operating system [11], and demonstrate its improved performance over existing schedulers on real applications.

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 current operating systems are designed for:

- *Soft real-time constraints.* Real-time applications have application-specific timing requirements that need to be met [31]. 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.
- *Insatiable resource demands and frequent overload.* Multimedia applications present practically an insatiable demand for resources. Today, video playback windows are typically tiny at full display rate because of insufficient processor cycles to keep up at full resolution. As applications such as real-time video are highly resource intensive and can consume the resources of an entire machine, resources are commonly overloaded, with resource demand exceeding its availability.
- *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 [32]. 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.
- *Co-existence with conventional computations*. Real-time applications must share the desktop with already existing conventional applications, such as word processors, compilers,

etc. 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 together 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. SMART is unique in its ability to automatically shed real-time tasks and regulate their execution rates 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) [7]. SMART then uses an urgency mechanism based on earliest-deadline scheduling [26] 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 some 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 [12], and WFQ, which has been the subject of much attention in current research [2, 7, 33, 38, 40]. 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 the algorithm and shows how the algorithm delivers behavior consistent with its principles of operations. Section 5 provides a comparison with related work. Section 6 presents a set of experimental results, followed by some concluding remarks.

2 The SMART interface and usage model

The SMART interface provides to the application developer *time constraints* and *notifications* for supporting applications with real-time computations, and provides to the user of applications *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 [30].

Multimedia application developers are faced with the problem of writing applications with time constraints. They typically know the deadlines that must be met in these applications and know how to allow these applications to 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 under UNIX, an application can tell the scheduler to delay a computation by "sleeping" for a duration of time. An application can also obtain simple timing information such as elapsed wall clock time and accumulated execution time. However, it cannot ask the scheduler to complete a computation before a given deadline, nor can it ask the scheduler whether or not it is possible for the computation to complete before a given deadline. The lack of system support exacerbates the difficulty of writing applications with time constraints and results in poor application performance.

By providing explicit time constraints, SMART allows applications to communicate their timing requirements to the system. 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 with time constraints and blocks of code that do not, thereby allowing application developers to freely mix real-time and conventional computations.

SMART also provides a simple upcall from the scheduler that informs the application that its deadline cannot be met. This upcall mechanism is called a notification. It 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

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

Table 1: Categories of applications

policies for deciding what to do when a deadline is missed. For example, upon notification, the application may choose to discard the current computation, perform only a portion of the computation, or change the time constraints of the computation. This feedback from the system enables adaptive real-time applications to degrade gracefully.

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 time constraints. As an example, we describe an audio/video application that was programmed using time constraints in Section 6.1.

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. These parameters can be used to bias the allocation of resources to provide the best performance for those applications which are currently 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. 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. The notions of priority and share apply uniformly to both real-time and conventional applications. This level of predictable control is unlike current practice, as typified by UNIX time-sharing, in which all that a user is given is a "nice" knob [39] whose setting is poorly correlated to the scheduler's externally observable behavior [29].

Our expectation is that most users will run the applications in the default priority level with equal shares. This is the system default and requires no user parameters. The user may wish to adjust the proportion of shares between the applications occasionally. A simple graphical interface can be provided to make the adjustment as simple and intuitive as adjusting the volume of a television or the balance of a stereo output. The user may want to use the priority to handle specific circumstances. Suppose we wish to run the PointCast application [34] in the background only if the system is not busy; this can be achieved simply by running PointCast with a low priority.

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.
- *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 would 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. This is the same motivation behind the design of multi-level feedback schedulers [23] 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

As summarized in Table 1, real-time and conventional applications have very diverse characteristics. It is this diversity that makes devising an integrated scheduling algorithm difficult. A real-time scheduler uses real-time constraints to determine the execution order, but conventional 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 which 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.

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 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 [27] 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 with the highest value-tuple. The algorithm attempts to schedule the candidate into a working schedule which is initially empty. The candidate is inserted in deadline order in this schedule provided its execution does not cause any of the 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 [2, 7, 33, 38, 40]. 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, weighted fair queueing executes the task with the earliest virtual finishing time to provide proportional sharing. SMART only uses the biased virtual finishing time in the selection of the candidates for scheduling, and real-time constraints are also considered in the choice of the application to run. This modification enables SMART to handle applications with aperiodic constraints and overloaded conditions.

Our algorithm organizes all the 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

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

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

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

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

Previous work in the domain of packet switching provides a theoretical basis for using the difference between the virtual time of a task and the queue virtual time as a measure of whether the respective task has consumed its proportional allocation of resources [7, 33]. 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 relative 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 the its corresponding queue. Suppose the task has time quantum Q, then its BVFT is

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

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 τ be the time a task's BVFT was last changed. Then, the task's BVFT is

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

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 [39]. 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 which 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 which 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

$$\Delta = v(t) - V_P(t), \quad (5)$$

where

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

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

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

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:

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

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

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

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.

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 N_R from growing large. For large N, the cost of scheduling with SMART in practice is expected to be similar to WFQ.

A more complicated algorithm can be used to reduce the complexity of managing the working schedule. In this case, a new working schedule can be incrementally built from the existing working schedule as new 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

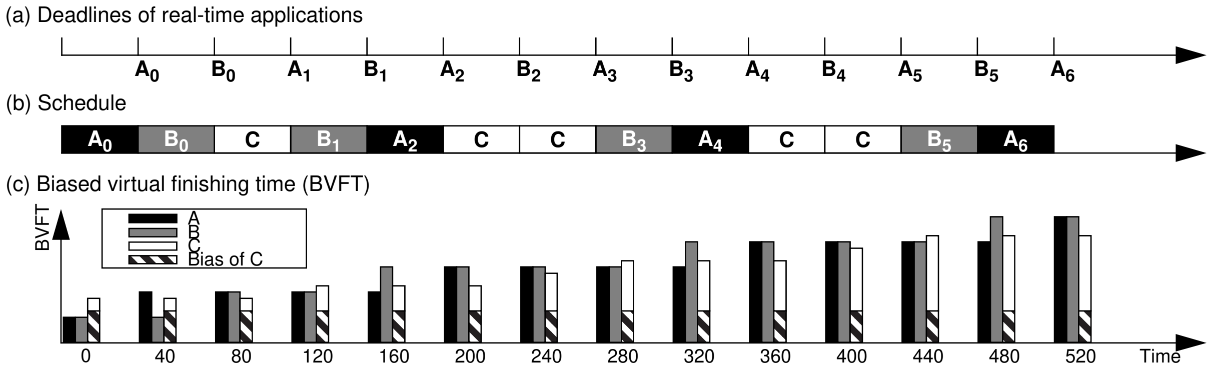

Figure 1: Example illustrating the behavior of SMART

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. Figure 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 BVFT are kept close together.

This example illustrates several important characteristics of SMART. First, SMART implements proportional sharing properly. In the steady state, *C* is given twice as much resources as *A* or *B*, which reflects the ratio of shares given to the applications. Second, the bias allows better response in temporary overload, but it does not reduce the proportional share given to the biased 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_2 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.

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 1, interactive and real-time applications may not use up all the resources that they are entitled to. Any unused cycles are proportionally distributed among those applications that can consume the cycles.

4.2.1 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 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 leftover 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 [26], which has been shown to satisfy all scheduling constraints, periodic or aperiodic, optimally [8].

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, and hierarchical scheduling.

5.1 Real-time scheduling

Real-time schedulers such as rate-monotonic scheduling [24, 26] and earliest-deadline scheduling [8, 26] are designed to make better use of hardware resources in meeting real-time requirements. In particular, earliest-deadline scheduling is optimal in underload. However, they do not perform well when the system is overloaded, nor are they designed to support conventional applications.

Resource reservations are commonly combined with real-time scheduling in an attempt to run real-time tasks with conventional tasks [5, 22, 25, 28]. 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 [5, 25] take advantage of earliest-deadline scheduling to provide optimal real-time performance in underload, the rate monotonic utilization bound used in [28] and the time interval assignment used in Rialto [22] are not optimal, resulting in lower performance than earliest-deadline approaches. In contrast with SMART, these approaches are more restrictive, especially in the level of control provided for conventional tasks. They do not provide a common mechanism for sharing resources across real-time and conventional tasks. In particular, with conventional tasks being given leftover processing time, their potential starvation is a problem. This problem is exacerbated in Rialto [22] in which even in the absence of reservations, applications with time constraints buried in their source code are given priority over conventional applications [21].

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

Unlike reservation mechanisms, best-effort real-time scheduling [27] 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 [1, 16].

5.2 Fair queueing

Fair queueing provides a mechanism which allocates resources to tasks in proportion to their shares. It was first proposed for network packet scheduling in [7], with a more extensive analysis provided in [33], and later applied to processor scheduling in [40] as stride scheduling. Recent variants [2, 38] provide more accurate proportional sharing at the expense of additional scheduling overhead. The share used with fair queueing can be assigned in accordance with user desired allocations [40], or it can be assigned based on the task's resource requirement to provide resource reservations [33, 38]. When used to provide reservations, an admission control policy is also used.

When shares are assigned based on user desired allocations, fair queueing provides more accurate proportional sharing for conventional tasks than previous fair-share schedulers [9, 19]. However, it performs poorly for real-time tasks because it does not account for their time constraints. In underload, time constraints are unnecessarily missed. In overload, all tasks are proportionally late, potentially missing all time constraints.

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

Unlike real-time reservation schedulers, fair queueing can integrate reservation support for real-time tasks with proportional sharing for conventional tasks [38]. However, shares for real-time applications must then be assigned based on their resource requirements; they cannot be assigned based on user desired allocations.

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

5.3 Hierarchical scheduling

Because creating a single scheduler to service both real-time and conventional resource requirements has proven difficult, a number of hybrid schemes [3, 6, 15, 16, 39] have been proposed. These approaches attempt to avoid the problem by having statically separate scheduling policies for real-time and conventional applications, respectively. The policies are combined using either priorities [6, 15, 39] or proportional sharing [3, 16, 18] as the base level scheduling mechanism.

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 [29].

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. Real-time applications will not take over the machine, but they also cannot meet their time constraints effectively as a result of the underlying proportional share mechanism taking the resource away from the real-time scheduler at an inopportune and unexpected time in the name of fairness [17].

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

SMART behaves like a real-time scheduler when scheduling only real-time requests and behaves like a conventional scheduler when scheduling only conventional requests. However, it combines these two dimensions in a dynamically integrated way that fully accounts for real-time requirements. SMART ensures that more important 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 Experimental results

We have implemented SMART in Solaris 2.5.1, the current release of Sun Microsystems's UNIX operating system. To demonstrate its effectiveness, 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 (SVR4-RT) and time-sharing (SVR4-TS) policies, and a WFQ processor scheduler. 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 [35] 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 6.1. 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 monaural samples. 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.

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 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. To allow a much finer resolution for scheduling events, we added a high res-

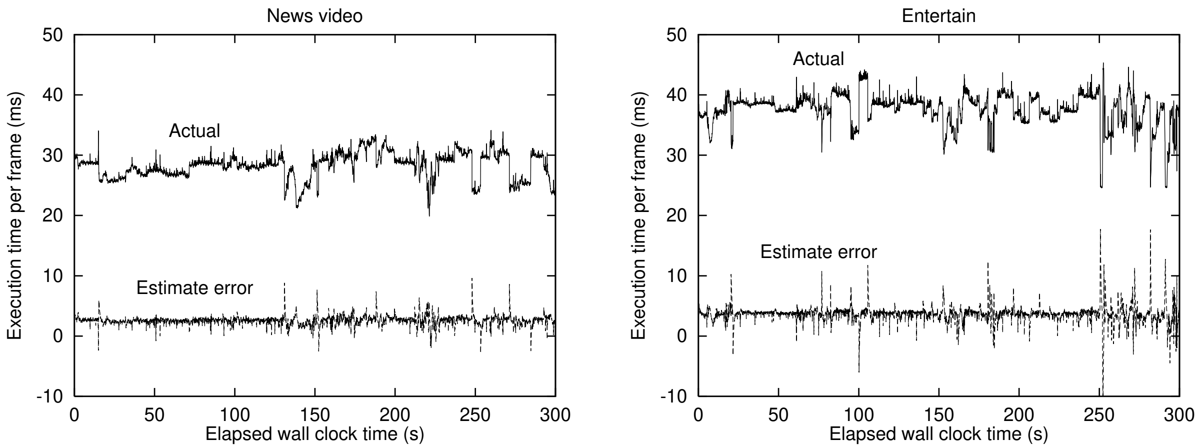

Figure 2: Actual vs. estimated execution time per JPEG image

olution timeout mechanism to the kernel and reduced the time scale at which timer based interrupts can occur. The exact resolution allowed is hardware dependent. On the testbed workstation used for these experiments, the resolution is 1 ms. The high resolution timing functionality was used for all of the schedulers to ensure a fair comparison.

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

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

6.1 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.

6.1.1 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 [13] 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 replace the application's discard mechanism with simply 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 2 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, the

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

Table 2: Standalone execution times of applications

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

Table 3: Standalone application quality metric performance

average execution time across all frames, and 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. Using the upper bound on the processing time as an estimate may yield a low utilization of resources; using the average processing time may cause too many deadlines to be missed.

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

6.2 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 2 shows the execution time of each application on an otherwise quiescent system using the UNIX SVR4 scheduler, measured over a time period of 300 seconds. We note that there is no significant difference between the performance of different schedulers when running only one application. The execution times include user time and system time spent on behalf of an application. The *Dhrystone* batch application can run whenever the processor is available and can thus fully utilize the processor. The execution of other system functions (fsflush, window system, etc.) takes less than 1% of the CPU time. The measurements on the real-time applications are taken every frame, and those for *Typing* are taken every character. None of the real-time and interactive applications can take up the whole machine on its own, with both *News* audio and *Typing* taking hardly any time at all. The video for *News* takes up 42% of the CPU, whereas *Entertain*, which displays scaled video, takes up almost 60% of the processor time.

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 [37]. As such, we measured the *Typing* character latency and determine the percentage of characters processed with latency less than 50 ms, with latency between 50-150 ms, and with latency greater than 150 ms. For *News* audio, it is desirable not to have any artifacts in audio output. As such, we measured the number of *News* audio samples dropped. For *News* video and *Entertain*, it is desirable to minimize the difference between the desired display time and the actual display time, while maximizing the number of frames that are displayed within their time constraints. As such, we measured the percentage of *News* and *Entertain* video frames that were displayed on time (displayed within 20 ms of the desired time), displayed early, displayed late, and the percentage of frames dropped not displayed. Finally, for batch applications such as *Dhrystone*, it is desirable to maximize the processing time devoted to the application to ensure as rapid forward progress as possible. As such, we simply measured the CPU time *Dhrystone* accumulated. To establish a baseline performance, Table 3 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 [36]. External versus internal measurements differed by less than 10 ms. The difference is due to the refresh time of the frame buffer.

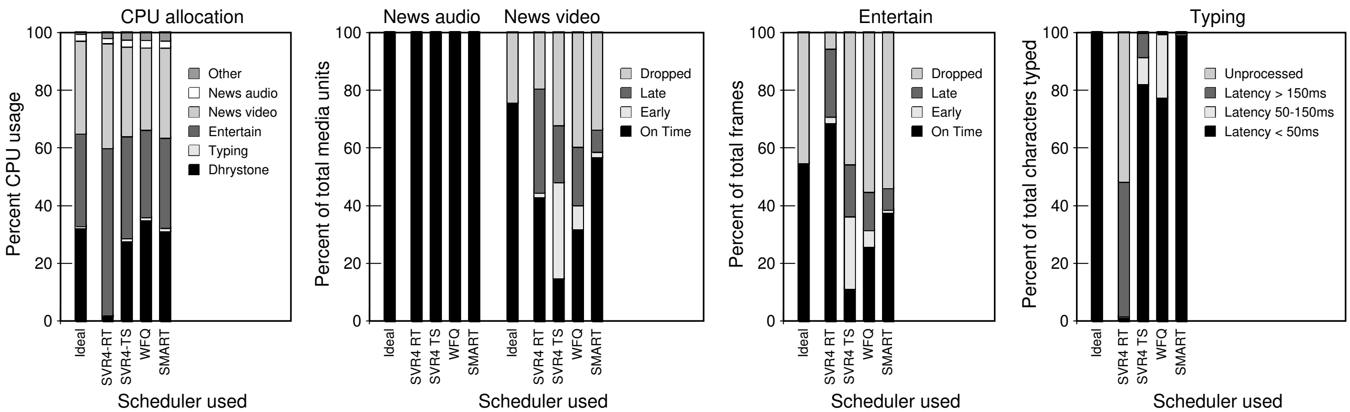

Figure 3: Comparison of scheduler application performance

For *Typing*, we measured the end-to-end character latency from the arrival of the character to the system in the input device driver, through the processing of the character by the application, until the actual display of the character by the X window system character display routine.

6.3 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 due to there being no more memory to allocate to the applications.

The UNIX SVR4 context switch time essentially measures the context switch overhead for a scheduler that takes almost no time to decide what 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.

6.4 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*. Even with an ideal scheduler, we expect the percentages of the frames dropped to be 25% and 45% for *News* video and *Entertain*, respectively.

Figure 3 presents the CPU allocation across different applications by different schedulers. It includes the percentage of the CPU used for executing other system functions such as the window system (labeled *Other*). The figure also includes the expected result of an ideal scheduler for comparison purposes. For the real-time applications, 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 4 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. Thus we will only concentrate on discussing the quality of the rest of the applications.

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

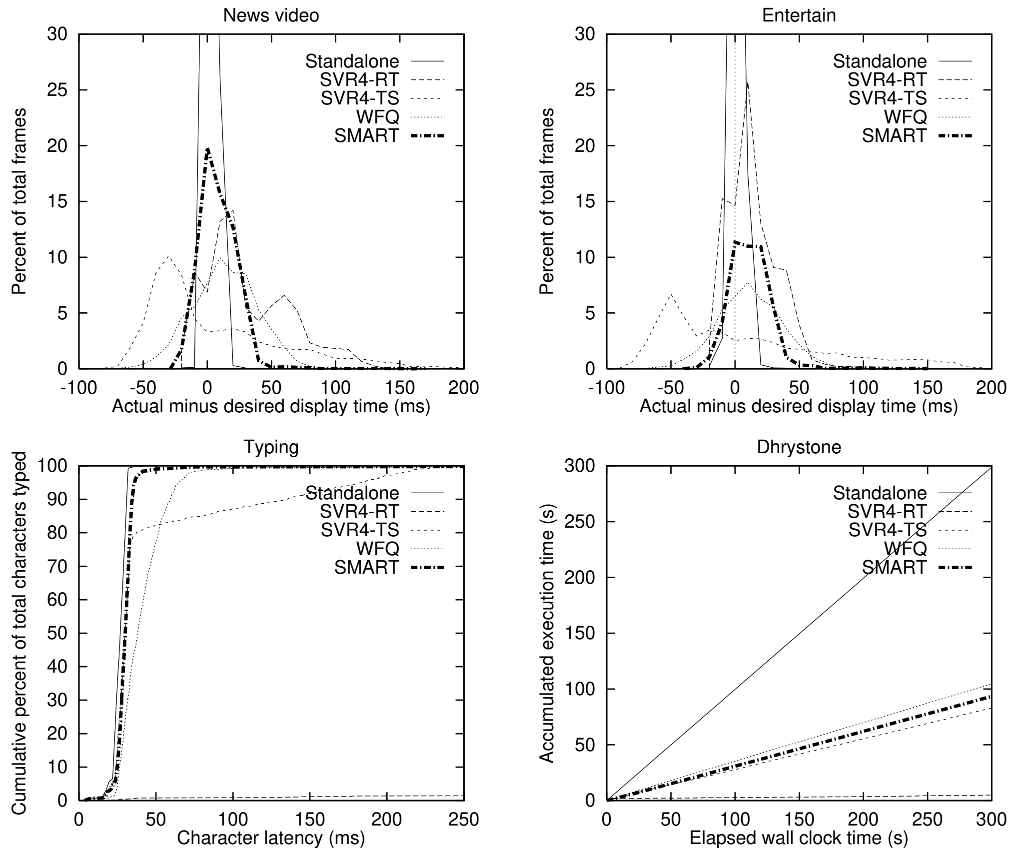

Figure 4: Distributions of quality metrics

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 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 in the number of frames dropped between the schedulers. We found that in each instance the scheduler drops about 6-7% more frames than the ideal computed using average computation times and the scheduler's specific allocation for the application.

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 4, 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, the WFQ delivers 32% and 26% of the total frames on time, which represents only 53% and 58% of the frames processed. There are many more late frames in the WFQ case than in SMART. The tardiness causes the applications to initiate the processing earlier, thus resulting in a correspondingly larger number of early frames. The 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 4, SMART is superior to both SVRT-TS and WFQ in handling the *Typing* application. SMART has the least average and standard deviation in character latency and completes the most number of characters in less than 50 ms, the threshold of human detectable delay.

While both SMART and WFQ deliver acceptable interactive performance, *Typing* performs worse with WFQ because a task does not accumulate any credit at all when it sleeps. We performed

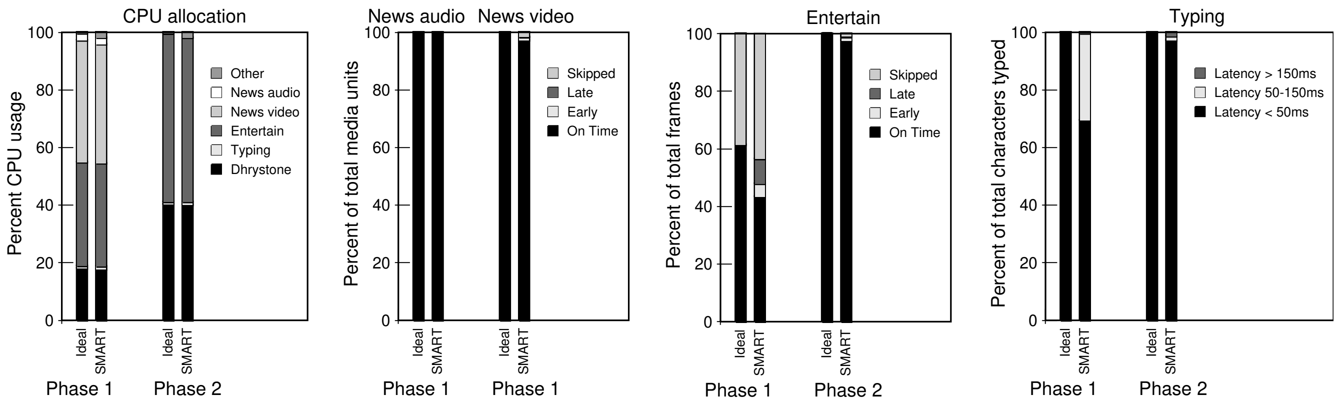

Figure 5: SMART application performance under a changing load when using end user controls

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.

6.5 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 is 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 5 shows the CPU allocation and quality metrics of the different applications run under SMART as well as an ideal scheduler. (Distributions of the data points are not included here due to lack of space.) 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 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 [4].

7 Concluding remarks

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

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 Jim, Duane, Amy Lim, Mendel Rosenblum, Alice Yu, Rich Draves, and the conference referees for helpful comments on earlier drafts of this paper. This work was supported in part by an NSF Young Investigator Award and Sun Microsystems Laboratories.

References

1. V. Baiceanu, C. Cowan, D. McNamee, C. Pu, J. Walpole, "Multimedia Applications Require Adaptive CPU Scheduling", Proceedings of the *IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems*, Washington, DC, Dec. 1996.
2. J. C. R. Bennett, H. Zhang, "WF^2Q: Worst-case Fair Weighted Fair Queueing", *IEEE INFOCOM '96*, San Francisco, CA, pp. 120-128, Mar. 1996.
3. G. Bollella, K. Jeffay, "Support for Real-Time Computing Within General Purpose Operating Systems: Supporting Co-Resident Operating Systems", *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Chicago, IL, pp. 4-14, May 1995.
4. S. K. Card, T. P. Moran, A. Newell, *The Psychology of Human-Computer Interaction*, L. Erlbaum Associates, Hillsdale, NJ, 1983.
5. G. Coulson, A. Campbell, P. Robin, G. Blair, M. Papathomas, D. Hutchinson, "The Design of a QoS Controlled ATM Based Communications System in Chorus", *IEEE JSAC*, 13(4), pp. 686-699, May 1995.
6. H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.
7. A. Demers, S. Keshav, S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm", *Proceedings of SIGCOMM '89*, pp. 1-12, Sept. 1989.
8. M. Dertouzos, "Control Robotics: The Procedural Control of Physical Processors", *Proceedings of the IFIP Congress*, Stockholm, Sweden, pp. 807-813, Aug. 1974.
9. R. B. Essick, "An Event-based Fair Share Scheduler", *Proceedings of the 1990 Winter USENIX Conference*, Washington, DC, pp. 147-161, Jan. 1990.
10. S. Evans, K. Clarke, D. Singleton, B. Smaalders, "Optimizing Unix Resource Scheduling for User Interaction", *Proceedings of the 1993 Summer USENIX Conference*, Cincinnati, OH, pp. 205-218, June 1993.
11. J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams, "Beyond Multiprocessing...Multithreading the SunOS Kernel", *Proceedings of the 1992 Summer USENIX Conference*, San Antonio, TX, pp. 11-18, June 1992.
12. P. Ffoulkes, D. Wikler, "Workstations Worldwide Market Segmentation", *Advanced Desktops and Workstations Worldwide*, Dataquest, June 1997.
13. N. G. Fosback, *Stock Market Logic*, Institute for Econometric Research, Ft. Lauderdale, FL, 1976.
14. L. Georgiadis, R. Guérin, V. Peris, K. N. Sivarajan, "Efficient Network QoS Provisioning Based on per Node Traffic Shaping", *IEEE/ACM Transactions on Networking*, 4(4), pp. 482-501, Aug. 1996.
15. D. B. Golub, "Operating System Support for Coexistence of Real-Time and Conventional Scheduling", Technical Report CMU-CS-94-212, School of Computer Science, Carnegie Mellon University, Nov. 1994.
16. P. Goyal, X. Guo, H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems", *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, pp. 107-122, Oct. 1996.
17. P. Goyal, Panel talk at the *IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems*, Washington, DC, Dec. 1996.
18. J. G. Hanko, "A New Framework for Processor Scheduling in UNIX", Abstract talk at the *Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Lancaster, U. K., Nov. 1993.
19. G. J. Henry, "The Fair Share Scheduler", AT&T Bell Laboratories Technical Journal, 63(8), pp. 1845-1858, Oct. 1984.
20. *IEEE Micro*, 15(4), Aug. 1996.
21. M. B. Jones, personal communication, July 1997.
22. M. B. Jones, D. Rosu, M-C. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, Oct. 1997.
23. S. J. Leffler, M. K. McKusick, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.
24. J. Lehoczky, L. Sha, Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166-171, Dec. 1989.
25. I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", *IEEE JSAC*, 14(7), pp. 1280-1297, Sept. 1996.
26. C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *JACM*, 20(1), pp. 46-61, Jan. 1973.
27. C. D. Locke, "Best-Effort Decision Making for Real-Time Scheduling", Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, May 1986.
28. C. W. Mercer, S. Savage, H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications", *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, pp. 90-99, May 1994.
29. J. Nieh, J. G. Hanko, J. D. Northcutt, G. A. Wall, "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications", *Proceedings of the Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Lancaster, U. K., pp. 35-48, Nov. 1993.
30. J. Nieh, M. S. Lam, "SMART UNIX SVR4 Support for Multimedia Applications", *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Ottawa, Canada, pp. 404-414, June 1997.
31. J. D. Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*, Academic Press, Boston, MA, 1987.
32. J. D. Northcutt, E. M. Kuerner, "System Support for Time-Critical Applications", *Proceedings of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Lecture Notes in Computer Science, Vol. 614, Heidelberg, Germany, pp. 242-254, Nov. 1991.
33. A. K. Parekh, R. G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case", *IEEE/ACM Transactions on Networking*, pp. 344-357, June 1993.
34. "PointCast Unveils First News Network that Reaches Viewers at Their Desktops", Press Release, PointCast Inc., San Francisco, CA, Feb. 13, 1996.
35. R. W. Scheifler, J. Gettys, "The X Window System, *ACM Transactions on Graphics*, 5(2), pp. 79-109, Apr. 1986.
36. B. K. Schmidt, "A Method and Apparatus for Measuring Media Synchronization", *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Durham, NH, pp. 203-214, Apr. 1995.
37. B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2nd ed., Addison-Wesley, Reading, MA, 1992.
38. I. Stoica, H. Abdel-Wahab, K. Jeffay, "On the Duality between Resource Reservation and Proportional Share Resource Allocation", *Multimedia Computing and Networking Proceedings, SPIE Proceedings Series*, Vol. 3020, San Jose, CA, pp. 207-214, Feb. 1997.
39. *UNIX System V Release 4 Internals Student Guide*, Vol. I, Unit 2.4.2., AT&T, 1990.
40. C. A. Waldspurger, "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management", Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.