The Design of SMART: A Scheduler for Multimedia Applications

Jason Nieh and Monica S. Lam
{nieh,lam}@cs.stanford.edu

Technical Report CSL-TR-96-697
Computer Systems Laboratory
Stanford University
June 1996

Abstract

We have created SMART, a Scheduler for Multimedia And Real-Time applications. SMART supports both real-time and conventional computations and provides flexible and accurate control over the sharing of processor time. SMART is able to satisfy real-time constraints in an optimal manner and provide proportional sharing across all real-time and conventional tasks. Furthermore, when not all real-time constraints can be met, SMART satisfies each real-time task's proportional share of deadlines, and adjusts its execution rate dynamically. This technique is especially important for multimedia applications that can operate at different rates depending on the loading condition. This paper presents the design of SMART and provides measured performance results of its effectiveness based on a prototype implementation in the Solaris ®operating system.

1 Introduction

Processor schedulers for multiprogrammed systems must multiplex shared resources in order to service the resource requests of concurrently running applications. These applications may be of different importance and may have different resource requirements. To be effective, the scheduler must account for these different importance and resource requirements. It should apply that information toward making the most efficient use of hardware resources in order to deliver the best application performance possible. In particular, when resources are scarce, it is crucial for the scheduler to provide flexible and accurate controls that can be used to provide better performance to those applications that matter most.

Traditional schedulers have evolved along a path that has emphasized notions of throughput and fairness. Their goal has been to effectively time-multiplex resources among conventional interactive and batch applications. However, there exists today a growing class of applications whose resource requirements are real-time in nature. Multimedia audio and video, virtual reality, transaction processing, and data acquisition are examples of this class of applications. Unlike conventional requests, real-time resource requests need to be completed within application-specific delay bounds, known as *deadlines*, in order to be of maximum value to the application [19]. Tardy results are often discarded as being of no value at all. To integrate these real-time applications in the workstation environment together with existing conventional applications, schedulers are faced with the question of how to service both real-time and conventional resource requests in order to allow them to co-exist and share resources effectively. Not only should it be possible for real-time applications to meet their deadlines, but conventional interactive applications should be able to obtain good responsiveness and conventional batch applications should be able to make reasonable forward progress, even in the presence of real-time requirements.

With the combined requirements of efficient resource utilization, support for applications of different importance, and support for real-time, interactive and batch resource requests, scheduling multimedia applications is a particularly challenging problem. While many schedulers have been proposed that satisfy some of these requirements in restricted application domains, effectively supporting the combination of these requirements in the general-purpose workstation environment is beyond the scope of current schedulers [18]. However, there are three useful concepts that have been previously identified independently for their effectiveness in restricted application domains: best-effort real-time decision making [2][15][26], exploiting the ability of batch applications to tolerate latency [1][10][11], and proportional sharing [3][7][20][23][24][25].

Best-effort real-time decision making (BE) combines earliest-deadline scheduling [14] with a unique priority for each request to provide optimal performance in *underload* and graceful degradation in *overload*. (A set of real-time resource requests is defined to be in underload if there exists a schedule that can meet all deadlines, and is defined to be in overload if no such schedule exists.) BE provides optimal performance in underload in that it can always satisfy all deadlines if at all possible with any scheduler, thereby making the most efficient use of hardware resources. When not all deadlines can be met, rather than making everyone late and delivering tardy results that may be of no value, BE provides graceful degradation by denying service to lower priority requests to ensure that higher priority requests can still meet their deadlines. However, BE does not support proportional sharing and requires that all requests have deadlines. As a consequence, it lacks even the ability to support conventional applications in a fair manner.

Multilevel-feedback schedulers, as typified by UNIX® time-sharing [1][11], take advantage of the ability of long running batch applications to tolerate longer and more varied service delays to deliver better response time to short interactive requests while attempting to ensure that batch applications make reasonable progress. We introduce the term *latency tolerance* as a measure of the ability of an application to tolerate latency in the servicing of its resource requests. Interactive and batch applications have very different latency tolerances. Interactive applications are often short and intermittent and desire fast response time. Their latency tolerances are typically small because service delays of even a couple seconds are often quite noticeable in these applications. On the other hand, batch applications are typically long running and less sensitive to service delays. Their latency tolerances are often large because even service delays of a few minutes make little difference if the result of the application takes several hours to compute in the first place. For example, a delay of a few seconds is quite noticeable when text editing, but it typically makes little difference whether a long compilation completes in four hours, or it completes in four hours and five minutes. However, while the idea of latency tolerance is attractive, the multilevel-feedback mechanisms used to achieve it are poorly understood [24] and are not amenable to predictable controls, as evident with the *nice* knobs provided with UNIX schedulers [17].

Proportional sharing, also known as weighted fairness, has long been advocated as an effective basis for allocating resources among competing applications. The principle is to allocate resources among competing applications in proportion to their relative weightings, which are sometimes called *shares*. For example, given two applications with shares 1 and 2 respectively, the second application would be allowed to consume twice as much of the resource as the first application. Shares provide a simple abstraction that can be used to bias the allocation of resources based on the importance of an application. However, deadlines and latency tolerance are unaccounted for in weighted fairness, and there are no known mechanisms that allow weighted fairness to be obtained with real-time requests in overload, when the importance of an application matters the most.

To provide effective support for multimedia applications, we have created SMART, Scheduler for Multimedia And Real-Time. SMART combines the advantages of best-effort real-time decision making, latency tolerance, and weighted fairness. SMART uses an algorithm based on weighted fair queueing (WFQ) [3] to implement weighted fairness. The concept of *virtual time* is used to measure the degree to which each task has consumed its proportional share of resources. Taking advantage of the ability of batch applications to tolerate longer and more varied service delays, we bias the virtual time with a notion of latency tolerance to provide better responsiveness to real-time and short interactive resource requests. The biased virtual time is then used as priority in a best-effort scheduling algorithm so as to satisfy as many real-time constraints as possible.

When scheduling real-time applications, SMART has the desirable behavior of a real-time scheduler: it ensures that deadlines are met whenever possible, and allows applications to degrade proportionally according to their relative importance when resources are insufficient to satisfy all resource requirements. When scheduling conventional applications, SMART has the desirable behavior of a conventional scheduler: it ensures that interactive computations deliver good responsiveness and batch computations make long term progress fairly according to their shares. More importantly, not only does our unified scheduler handle each class of computation effectively, it also handles the combination of all classes of computations seamlessly. We have implemented a SMART processor scheduler in the Solaris ®operating system to demonstrate its effectiveness.

This paper presents the design of SMART, and focuses on the aspects of its design that enable it to provide proportional sharing among both real-time and conventional applications. The paper is organized as follows. Section 2 presents an overview of our approach and describes our assumptions and scheduling model. Section 3 describes the mechanisms used in SMART to support weighted fairness. Section 4 describes the mechanisms used in SMART to take advantage of latency tolerance. Section 5 describes the mechanisms used in SMART to provide best-effort scheduling of real-time applications while allowing conventional applications to co-exist and share resources effectively. Section 6 compares SMART with other scheduling schemes. Section 7 presents performance measurements based on our Solaris implementation prototype of SMART. Section 8 presents some concluding remarks.

2 Approach

2.1 Assumptions and Context

SMART is designed to provide effective resource management in the workstation environment. In this environment, good price performance is critical and as a result, being able to deliver efficient processor utilization is imperative. Unlike single function systems such as embedded systems or video game machines, the workstation is a multiprogrammed environment that must be able to support the needs of concurrently running applications. As is typical in such environments, applications can be preemptively multitasked by the scheduler as needed to meet their requirements. Note that this environment is a dynamic one in which the mix of applications that need to be executed may change at any time. As a result, scheduling decisions must be done on-line in order for the scheduler to effectively manage resources.

From the perspective of the real-time systems domain, our focus here is on scheduling soft real-time applications in which missed deadlines are undesirable but not catastrophic. Many applications, including those in multimedia, fall in this category of real-time applications. These applications are able to adapt to overloaded conditions that result in their deadlines not being met. Applications such as video playback are amenable to dropping video frames as a means of degrading gracefully in overloaded conditions. Their primary requirement is not simply that all deadlines must be met. Instead, as many deadlines as possible should be met, and when deadlines cannot be met, resources should be provided in a predictable manner that allows the applications to degrade gracefully. Rather than making everybody equally late, the scheduler should ensure that as little processing time as possible is wasted on tardy results that would otherwise be discarded when the processing time can be used instead to service requests whose deadlines can be met.

On the other hand, in hard real-time applications, a missed deadline is defined to be catastrophic. These applications run in fault-tolerant environments with excess resources, in contrast to commercial workstation environments, where price performance issues matter. We note that SMART also provides features, such as combining user-defined priorities with earliest-deadline scheduling, for supporting these kinds of applications that are of strictly greater importance, as opposed to just proportionally more important. However, a discussion of these additional features is beyond the scope of this paper.

Implicit in the following discussion is the fact that SMART provides a rich interface that facilitates the cooperation of applications and users with the scheduler to allow the flow of information necessary to manage resources effectively. In particular, the interface allows applications and users to specify their importance and real-time requirements, and allows applications to be notified via an upcall mechanism when their deadlines cannot be met so that they are able to adapt to the available resources. While a detailed account of the design and rationale of the SMART interface is beyond the scope of this paper, a brief summary of some of the relevant scheduling attributes is presented in Section 2.2.

2.2 Task Model and Definitions

We briefly describe the execution model common in these environments as a basis for describing our approach. In this paper, we use the term *task* to refer to a thread of execution. The time from when a task is created until when a task is completed is called the *lifetime* of the task. During the lifetime of a task, a task may be in one of two states: *runnable* and *blocked*. A task is runnable if it is running or is available to be run on the processor. A task is blocked if it is waiting for an event to occur, such as I/O, before it can run. A task that has been blocked for a long time, for example while waiting for user's input, is considered to be *inactive*; otherwise, it is considered *active*.

A task is made up of a sequence of resource requests, each of which may have real-time or conventional requirements. The resource requests of a task are defined to be serialized; a task cannot have multiple resource requests at once for a given resource. As previously described, the basic difference between real-time and conventional resource requests is that real-time requests have explicit deadlines by which they desire to complete while conventional requests do not have explicit deadlines. A request is said to be more *urgent* if it has an earlier deadline. Since conventional requests do not have deadlines, real-time requests are said to be more urgent than conventional requests. Note that resource requirements may change during the lifetime of the task. We shall refer to a task as a real-time task if it has real-time requirements at the time that it is being scheduled. Similarly, we shall refer to a task as a conventional task if it has conventional requirements at the time that it is being scheduled.

For the purposes of this paper, there are four primary scheduling attributes associated with each task: *deadline*, *time-quantum*, *latency tolerance*, and *share*. As previously discussed, the deadline specifies when the respective resource request should be completed; conventional tasks have no deadlines. The time-quantum specifies the processing time required to complete the resource request. The latency tolerance indicates the ability of the task to tolerate latency in the servicing of its resource requests. The share specifies the relative proportion of resources that can be allocated to a task.

2.3 Algorithm Overview

The basic SMART scheduling algorithm combines the concepts of proportional sharing, latency tolerance, and best-effort real-time decision making. It seeks to optimize not just how much processing time each task is allocated, but also when each task is given its allocation of processing time.

Our design uses an algorithm based on weighted fair queueing (WFQ) [3] to implement proportional sharing. We use the concept of *virtual time* to measure the degree to which each task has consumed its proportional share of resources. We introduce the notion of latency tolerance to take advantage of the ability of batch applications to tolerate longer and more varied service delays, thereby providing better responsiveness to real-time and short interactive resource requests. A task's virtual time is adjusted according to its latency tolerance to determine its *virtual latency time*, which is used as a system-internal priority in a best-effort scheduler to schedule the resource requests. Conventional tasks do not have any artificial deadlines and are run only when they have been unduly delayed from receiving their proportional share of resources. That is, a conventional task will not make a real-time task miss its deadline unless the processor is overloaded for so long that the conventional task's time allocation falls below its tolerated level of unfairness. Consider first the case when all the real-time tasks request less than their proportional share. Whenever there are real-time tasks are present, SMART behaves like a simple BE scheduler by executing them in deadline order; otherwise, it runs the conventional tasks based on proportional sharing. On the other hand, when the system is overloaded, for real-time tasks requesting more resources than their proportional share, SMART will drop the appropriate real-time requests to dynamically adjust the rates of these tasks. We describe each of these components of the scheduling algorithm in more detail in the following sections.

3 Proportional Sharing

SMART allows the user to choose how the resources should be proportioned among all tasks, be they real-time or conventional, according to his preference. (The default is that all the tasks should share the resources equally). It should be possible for real-time applications to satisfy their deadlines at the expense of slower forward progress for conventional applications, if so desired. Similarly, it should also be possible for more important conventional applications to garner a larger portion of the resources even if it means not being able to satisfy the deadlines of less important real-time applications. In other words, whether a task has deadlines is independent from the proportion of the resources it should be allowed to consume. SMART separates the concepts of urgency from fairness, and can provide proportional sharing among and across real-time and conventional tasks.

We use a mechanism similar to WFQ to keep track of the resource consumption of each task in order to determine whether or not it has received more or less than its weighted fair portion of resources. For each task i, we define a *virtual time* $V_i(t)$ which advances at the rate of its execution time $E_i(t)$ divided by its share S_i: $V_i(t) = E_i(t)/S_i$. We also define a *global virtual time* $G_N(t)$ which advances at the rate of elapsed wall clock time divided by the total number of shares of all *active* tasks: $G_N(t) = t/\sum_{j=1}^{N} S_j$, where N is the number of active tasks. When a task is created, the virtual time of the task is set equal to the global virtual time.

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 global virtual time as a measure of whether the respective task has consumed its proportional allocation of resources [3][20]. In particular, if $V_i(t) = G_N(t)$, then $E_i(t) = t \times S_i / \sum_{j=1}^{N} S_j$. In other words, if the virtual time of a task is equal to the global virtual time, the task is considered to have received its proportional allocation of resources. Similarly, if the virtual time of a task is less than the global virtual time, the task is considered to have consumed less than its proportional allocation of resources. If the virtual time of a task is greater than the global virtual time, the task is considered to have consumed more than its proportional allocation of resources.

Since the global virtual time advances at the same rate for all tasks, the relative magnitudes of the virtual times provides a relative measure of the degree to which each task has received its proportional share of resources. In particular, for any two tasks *i* and *j*, if $V_i(t) = V_j(t)$, then $E_i(t)/E_j(t) = S_i/S_j$. In other words, if the virtual times of two tasks are equal, then their relative execution times are proportional to their shares. Similarly, if one task has a virtual time that is less than that of another, then the smaller virtual time task is considered to have consumed a lesser amount of its proportional share of resources than the larger virtual time task. One result of this is that a scheduler whose goal is to provide each task with its instantaneous proportional share could simply service tasks in order of their virtual times, as is done in [25].

The virtual time of each task advances at a rate that is independent of the shares of other tasks in the system. As a result, given the processing time required for a resource request, we can determine at the arrival of a task's resource request what the task's virtual time will be when its resource request is completed. This is called the *virtual finishing time* of the resource request, which we denote as $F_i(t)$ for a task *i*. For a task *i*, given a processing time requirement of Q_i, it follows that $F_i(t) = V_i(t) + Q_i/S_i$. We refer to the processing time required for a task's resource request as its *service-time*. For conventional tasks, their computations are typically divided into resource requests for time-slicing, each with a service-time equal to a scheduling time quantum on the order of 10-100 ms. When one time quantum is used up, a new time quantum is simply assigned. For real-time tasks, the resource requests correspond to the unit of computation that must complete before a deadline. The service-time of each resource request is equal to the estimated execution time required to complete the request before its deadline.

As we shall describe further in Section 5, the basic SMART scheduling algorithm gives preference to tasks with earlier virtual finishing times. Our choice of using virtual finishing times instead of perhaps the more obvious choice of using virtual times can be explained in part by comparing two scheduling algorithms, one that services tasks in order of increasing virtual times, which is known as Generalized Processor Sharing (GPS) [20] in the ideal case when tasks can be multiplexed at an infinitesimally fine granularity, and the other that services tasks in order of increasing virtual finishing times, which is known as WFQ as well as packet-by-packet GPS (PGPS) [20] in the case when requests are non-preemptible once they have started executing.

Although it is an idealized model, GPS provides instantaneous weighted fairness. It has been previously shown that WFQ provides a good approximation to GPS in providing weighted fairness [20]. Furthermore, the amount of time that a resource request is delayed when scheduled by WFQ as compared to GPS is bounded to be less than or equal to the product of the total number of shares of active tasks and the maximum resource request service-time. This bound however assumes that requests are non-preemptible once they have started executing, which is the case in the domain of packet switching in which WFQ was derived. However, for processor scheduling, resource requests are preemptible even after they have started executing. In this case, the delay bound is zero. In other words, when tasks are serviced in order of increasing virtual finishing times, a resource request is never completed later than it would be under GPS. Servicing tasks in order of increasing virtual finishing times results in better response time while still maintaining overall weighted fairness across multiple resource requests. We will discuss this issue further in Section 4 in conjunction with exploiting latency tolerance.

In addition, by ordering tasks based on virtual finishing times instead of virtual times, the relative ordering of tasks does not change at each moment, but only changes when the resource request of a task changes. For real-time tasks, this occurs at the arrival of a new resource request with a new deadline. For conventional tasks, this occurs at the completion of a resource request and the assignment of a new time quantum. This task ordering property is crucial for the execution of real-time resource requests because it allows their resource requests to be serviced as entities. As a result, once a real-time request has been determined to be able to complete before its deadline and has begun executing, it is possible to ensure that the request is not further delayed by other currently active tasks and can therefore complete before its deadline.

4 Latency Tolerance

As we have previously described, latency tolerance is a time unit measure of the amount of delay that a given task can tolerate in the servicing of its resource requests. Our previous discussion in Section 3 assumed that all tasks have zero latency tolerance. However, another benefit of the use of virtual finishing times in SMART to provide proportional sharing is that the virtual finishing times can be augmented to account for tasks that have different latency tolerances.

We have already pointed out that our use of virtual finishing times provides both weighted fairness and better response time than simple instantaneous weighted fairness of the kind provided by GPS. In particular, since the virtual finishing time of a task is based on the size of its resource request, tasks with shorter resource requests will initially obtain better response time than tasks with longer resource requests. To be more precise, among tasks with equal virtual times, those tasks with shorter resource requests will be given preference over those with longer resource requests, thereby obtaining better response time initially. However, if the tasks with shorter resource requests continue to consume resources, then since virtual times, and hence virtual finishing times, advance based on execution time, the longer resource requests will eventually be executed and obtain their proportional share of resources as well. The faster response time obtained by short resource requests is not at the expense of overall weighted fairness, but is only at the expense of instantaneous weighted fairness. Since applications at most care about weighted fairness at the granularity of their resource requests, attempting to provide instantaneous weighted fairness is actually of little value and hurts overall performance.

While real-time tasks in particular desire weighted fairness at the granularity of their resource requests, even this granularity of weighted fairness is not necessary for conventional tasks such as long running batch applications. While the resource request size of conventional tasks is based on a time quantum chosen by the scheduler that is typically on the order of 10-100 ms, long running batch applications for instance are often able to tolerate much longer delays in the servicing of their individual resource requests with no observable effect on their resulting performance.

For these conventional tasks, we introduce the idea of a *virtual latency tolerance*. The virtual latency tolerance is a virtual time measure of the ability of a task to tolerate longer and more varied service delays. Given the latency tolerance of a task, the virtual latency tolerance of the task is equal to its latency tolerance divided by its shares. Rather than just using virtual finishing times for ordering tasks to determine which tasks should given preference for being serviced, SMART orders tasks based on their virtual finishing times plus their respective virtual latency tolerances. The sum of the virtual finishing time and its corresponding virtual latency tolerance is called the *virtual latency time*. For real-time tasks, and conventional tasks with zero latency tolerance, the virtual latency time reduces to the virtual finishing time.

Note that the latency tolerance of a conventional task, and hence its virtual latency tolerance, need not be static values. Instead, the latency tolerance can be automatically adjusted by the system in a table-driven manner, similar to the way priorities and time-quanta are adjusted in UNIX SVR4 to implement time-sharing [1]. For instance, the latency tolerance can be allowed to increase if the task does not interact with the user for extended periods of time. The idea of a dynamically adjusted latency tolerance based on execution time is somewhat analogous to the idea of a decaying priority based on execution time which is used in multilevel-feedback. However, while multilevel-feedback affects the actual average amount of resources allocated to each task, latency tolerance only affects the response time of a task and does not affect its overall ability to obtain its proportional share of resources. By separating the mutually exclusive dimensions of weighted fairness and responsiveness, our use of latency tolerance is able to provide both weighted fairness and responsiveness in a systematic fashion.

5 Best-effort Scheduling

SMART considers a task with a smaller virtual latency time to have a higher priority (the term priority here refers to a system-internal attribute, not a user-defined parameter). Note that the priority is a dynamic attribute, as it reflects whether a task has been given its proportional share. SMART uses a best-effort scheduling algorithm to combine earliest-deadline scheduling with the priority values. Earliest-deadline scheduling is crucial to make efficient use of resources in meeting real-time requirements in underload. When not all real-time requirements can be met, our use of virtual latency times employs weighted fairness to determine when to shed real-time requests and scales back the resource consumption of tasks requesting more than their proportional share.

The first step in the scheduling algorithm is to create a priority list by sorting all active tasks in decreasing priority order. This sorting process is done on an incremental basis. Because of our use of virtual finishing times, the priority list only changes when a task's resource request changes. When a task's resource request changes, its position in the priority list must be updated. Since resource requests typically only change after a task has executed, only the position of the last executed task changes in the priority list. As a result, the priority list can be updated using a simple linear traversal to find the new position in the list of the last executed task. The complexity of this operation is $O(N)$, where N is the number of tasks in the priority list.

If the highest priority task has no deadline, for example, in the case of a conventional task, the scheduler simply executes the task with the highest priority. If there are real-time tasks in the system, this can potentially cause a real-time task with an urgent deadline to be shed. Note that since a conventional task's virtual latency time is biased by its latency tolerance, a conventional task will not make a real-time task miss its deadline unless the processor is overloaded for so long that the conventional task's time allocation falls below its tolerated level of unfairness.

Otherwise, SMART applies best-effort scheduling on all the real-time tasks with higher priorities than those without deadlines. In the degenerate case where there are no tasks without deadlines, it simply considers all the real-time tasks. SMART iteratively selects tasks among this set in decreasing priority order and inserts them into an initially empty schedule in increasing deadline order. The schedule defines an execution order for servicing the real-time resource requests, and is said to be *feasible* if the set of task resource requirements in the schedule can be completed before their respective deadlines if serviced in the order defined by the schedule. 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 schedule is feasible, let c_j be the service-time of task j, and e_j be the total time task j has already spent running toward completing the service-time. Let r_j be the percentage of the processor required by a periodic real-time task; r_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:

Resource requirement of task j: $R_j(t) = c_j - e_j + r_j \times (t - d_j), t \geq d_j$

A schedule S is then feasible if for each task i in the schedule with deadline d_i, the following inequality holds:

Feasibility test: $d_i \geq t + \sum_{j \in S | d_j \leq d_i} R_j(d_i), \dot{\forall} i \in S$

On each task insertion into the schedule, the resulting schedule that includes the newly inserted task is tested for feasibility. If the resulting schedule is not feasible, the newly inserted task is removed from the schedule. If the resulting 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 priority tasks are only inserted into the schedule if they do not cause any of the current and estimated future resource requests of higher priority tasks to miss their deadlines. The iterative selection process is repeated until SMART runs out of tasks or if 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 priority (earliest virtual latency time) 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. Finally, SMART notifies a task via an upcall mechanism if a task misses its deadline, and thereby allows the task to determine its own policy for handling the exception [5].

If all the real-time tasks request less than their proportional share, the scheduler will be able to run them in deadline order, and use the remaining time to run the conventional tasks. When the system is overloaded, for those real-time tasks that request more resources than their proportional share, SMART dynamically reduces their execution rate by simply shedding requests with low priority.

5.1 Examples

Let us consider a few simple examples to help illustrate the behavior of the SMART algorithm. We start off with two real-time tasks *R1* and *R2*. Task *R1* has a periodic deadline every 80 ms and requires 20 ms of processing time to satisfy each deadline. Task *R2* has a periodic deadline every 40 ms and requires 30 ms of processing time to satisfy each deadline. Given these two tasks, SMART will determine that both tasks can satisfy their respective deadlines and will execute them in earliest-deadline-first order, irrespective of what share each task is assigned. Note that we use periodic tasks here as examples for simplicity only. Neither task is required to be periodic in order for SMART to meet all of their deadlines.

Now let us add a conventional batch task *C1* to the task mix. By estimating the load on the system, SMART will determine that in the steady-state, it is not possible to allow task *C1* to make forward progress while allowing tasks *R1* and *R2* to meet all of their deadlines. The allocation of resources given to each task is then related to the share of each task. If tasks *R1*, *R2*, and *C1* have equal shares, then each task is allowed to use one-third of the machine. As a result, task *R1* will meet all of its deadlines because it only requires one-quarter of the machine to meet all of its deadlines. The remaining three-quarters of the machine will be equally distributed between tasks *R2* and *C1*. SMART would then service task *R2* in 30 ms quanta, allowing it to use its three-eighths of the machine to satisfy half of its deadlines while skipping the other half of its deadlines due to insufficient resources. Task *C1* would then consume the remaining three-eighths of the machine. On the other hand, if the shares for tasks *R1*, *R2*, and *C1* are 1, 3, and 4, respectively, then *R1* and *R2* will each be able to meet half of their deadlines while *C1* consumes half of the machine.

If task *C1* completes and is therefore removed from the task mix, SMART will determine that it is again possible to meet the deadlines of both tasks *R1* and *R2*. Tasks *R1* and *R2* will again execute in earliest-deadline-first order and meet all of their deadlines. Note that SMART handles these transitions between underload and overload without requiring any adjustments to the share of each task to achieve optimal underload performance and graceful degradation by proportional sharing in overload. This behavior is not possible with mechanisms based just on proportional sharing.

To help illustrate the effect of latency tolerance, let us consider the transient as opposed to steady-state behavior of the SMART algorithm. We start off with a long-running conventional batch task *C2* that has accumulated a latency tolerance of 100 ms and has a time-quantum of 10 ms. An aperiodic real-time task *R3* arrives requiring 33 ms of processing time before its 33 ms deadline. For simplicity, assume that both tasks have equal shares. At the arrival of task *R3*, both tasks will have the same virtual time. When task *R3* arrives, since task *C2* has a large enough latency tolerance, task *R3* will be able to run immediately and thereby complete before its deadline. After task *R3* completes, task *C2* will get to execute again.

If however task *R3* is a periodic real-time task that continues to requiring processing time to meet additional deadlines, then *R3* will only be allowed to run until task *C2* reaches its tolerated level of unfairness as measured by its latency tolerance. Assuming a periodic resource request of 33 ms for every 33 ms deadline, task *C2* with its 100 ms latency tolerance will allow task *R3* to meet three of its deadlines before it will run and cause *R3* to miss a deadline. Subsequently, *R3* will only be able to meet every other deadline in steady-state as *C2* consumes its fair proportion of the machine in accordance with its share. Note that simply using a larger time-quantum for *C2* does not result in the same desirable behavior for the given task mix.

5.2 Complexity

Our experience with an implementation of the algorithm indicates that the complexity of the overall algorithm in practice is dominated by the cost of insertion into the priority list, which is $O(N)$, where N is the number of tasks in the priority list. In the worst case, which occurs if each task needs to be selected and feasibility tested against all the other tasks in the schedule, the complexity of the algorithm is dominated by the complexity of the iterative selection process, which is $O(N^2)$, where N is the number of tasks in the priority list. Even with this complexity, simulation studies showed a marked improvement in the value the system delivered to the user under low-to-medium overhead assumptions [15].

However, it is unlikely for the worst case to occur in practice for any reasonably large N. The primary reason for this is because there are inevitably some conventional tasks in any workstation environment. These conventional tasks limit the number of tasks that need to be considered in the iterative selection process. Furthermore, real-time tasks typically have short deadlines so that if there are a large number of real-time tasks, the scheduler will determine that the schedule is no longer feasible before all of the tasks need to be individually tested for insertion feasibility.

6 Comparison with Other Schemes

6.1 Multilevel-feedback

Like multilevel-feedback schemes [1][10][11], SMART takes advantage of latency tolerance to improve the response time of applications that are more sensitive to delay. However, unlike multilevel-feedback, SMART provides a systematic approach to exploiting latency tolerance that can be combined with proportional sharing to provide predictable controls over the allocation of resources. As a result, SMART does not suffer from the non-intuitive behavior of the nice controls provided with multilevel-feedback schemes such as UNIX time-sharing, in which even small changes in the nice controls can lead to large and unpredictable effects. As previously demonstrated [17], nice controls require a great deal of experimentation in order to find a set of control values that work well for a given application mix, and the settings often only work for that exact application mix. Finally, while multilevel-feedback schemes cannot effectively schedule real-time applications, SMART provides the ability for latency tolerance to be exploited to improve the performance of real-time applications as well.

6.2 Proportional Sharing

Like proportional share schemes [3][7][20][23][24][25], SMART recognizes the need to provide flexible and accurate control over the allocation of resources for applications of different importance. SMART uses shares to provide a simple abstraction that can be used to bias the allocation of resources based on the importance of an application. However, unlike proportional share schemes, SMART provides an additional mechanism for accounting for the different latency as well as allocation requirements of real-time, interactive, and batch applications to make more efficient use of hardware resources while preserving the proportional share abstraction.

This results in three main differences between SMART and other proportional share mechanisms. First, in underload, SMART is optimal for both periodic and aperiodic real-time tasks. Other proportional share mechanisms are at best optimal for strictly periodic real-time tasks when their shares are set equal to their resource requirements.

Second, in overload, SMART is able avoid wasting processing time on deadlines that cannot be met and instead use each real-time task's proportional share of the machine toward meeting deadlines. Other proportional share mechanisms either result in all tasks being proportionally late, potentially missing all deadlines, or disallow proportional sharing in overload through first-come-first-serve admission control policies.

Third, SMART is able to transition effectively between underloaded and overloaded conditions. For example, it is possible in SMART to share resources fairly in overload while ensuring optimal performance in underload. Each task can simply be assigned an equal share. SMART estimates the load on the system to determine if all deadlines can be met. If that is possible, SMART meets all of the deadlines. Otherwise, it sheds tasks whose deadlines cannot be met based on their shares. As alluded to in the examples in Section 5.1, this kind of behavior is not possible with other proportional share mechanisms [3][7][20][23][24][25]. To get the desired overload behavior in these other schemes, the shares must be equal, but then deadlines will be missed in underload since the resource requirements may be different. If the shares are set according to the resource requirements, then the greedy tasks will monopolize the system in overload.

The problem is that other proportional share schemes only have one dimension one which to base their scheduling, which is the share of the task. Adding a QoS layer on top to adjust the shares depending on loading condition simply begs the question of how that adjustment is made, much in the same way that priorities can claim to be a complete scheduling solution if only one knew what priorities to assign. On the other hand, SMART has both shares and deadlines. It uses that information to estimate the load on the system to determine when all resource requirements can be met in spite of their shares, and when the shares are needed to ensure that greedy but unimportant tasks do not monopolize the machine. As a result, SMART is able to transition effectively among different loading conditions in a way not possible with just proportional sharing.

6.3 Real-time Scheduling

Real-time schedulers are predominantly based on either rate-monotonic [12][14][21] or earliest-deadline scheduling [14]. Rate-monotonic scheduling services periodic real-time tasks by shortest-period-first. It can be used in environments where only statically assignable priorities are available for scheduling, but at the cost of only being able to deliver 70% resource utilization in the worst-case. Earliest-deadline scheduling services real-time tasks in earliest-deadline order and is always able to deliver 100% resource utilization. However, the basic rate-monotonic and earliest-deadline schedulers are ill-equipped to handle resource requests that exceed their resource utilization bounds, although some ad-hoc "tricks" [22] have been used to address this issue.

Recognizing the need for scheduling support even when all real-time requirements cannot be met, BE [2][15] combined earliest-deadline scheduling with a unique priority for each request to provide optimal performance in *underload* and graceful degradation in *overload*. However, it is limited by the need for a unique priority and the lack of a practical method for providing them. Its use of priority is uncorrelated with resource consumption rates and thus it is unable to support proportional sharing. Furthermore, it does not handle periodic real-time tasks well in overload as it cannot anticipate future resource requests. SMART extends BE by addressing both of these issues as well as providing a unified mechanism for scheduling the combination of real-time and conventional tasks.

Several schemes for supporting multimedia applications [8][9][16] have evolved from real-time schedulers. These approaches do not allow conventional applications to function effectively in the presence of real-time applications. Each of these schemes rely on the use of artificial rates, artificial deadlines, first-come-first-serve resource reservations, or some combination of the above. In the presence of real-time applications, none of these approaches allow conventional applications to dynamically adapt to system loading conditions. In contrast, SMART provides proportional sharing of resources for conventional applications even in the presence of real-time resource requests so that both classes of applications can perform effectively.

6.4 Two-level Scheduling

Because creating a single scheduler to service both real-time and conventional resource requirements has proven difficult, a number of hybrid schemes [1][4][6] have been proposed based on the traditional two-level scheduler approach [13]. These approaches attempt to avoid the problem by having statically separate schedulers for real-time and conventional applications, respectively. However, all of the mechanisms that have been used for combining these scheduling policies ignore real-time requirements. In the best case, these approaches have the same limitations as conventional schedulers. In the worst case, pathological behaviors result in runaway real-time computations with the user being unable to regain control of the system [17]. 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 scheduling of actual processing time takes place.

Like two-level approaches, 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. In addition to allowing a wide range of behavior not possible with static schemes, SMART is better able to adapt to varying workloads and provide more efficient utilization of resources.

7 Implementation and Performance Results

We have implemented a prototype SMART processor scheduler in a pre-release build of the Solaris operating system, version 2.5.1. The scheduler handles the execution of all tasks in the system. To demonstrate its effectiveness, we present some measured performance results based on running a set of simple applications on an unoptimized version of the SMART scheduler prototype. The experiments described here were performed on a fully functional uniprocessor 150 MHz hyperSPARC™ SPARCstation™ 10 workstation. By a fully functional system, we mean that all experiments were performed with all system functions running and the system connected to the network. All measurements were performed using a minimally obtrusive tracing facility that logs events at significant points in both the application and system code. This is done via a light-weight mechanism that writes time-stamped event identifiers into a memory log. A suite of tools exist to post-process these event logs and obtain accurate representations of what happens in the actual system.

To provide a context for our experiments, we first describe some of the relevant system timing characteristics and system overheads in the workstation used to conduct our experiments. There are two timers of interest in the system, the hi-resolution timer and the clock interrupt timer. The hi-resolution timer is based on a half microsecond resolution hardware counter in SPARCstation 10 systems. Reading the counter value takes roughly one microsecond. This timer is ideally suited for cheap, accurate interval timing and is used for all of our timing measurements. The clock interrupt timer interrupts the processor once every 10 ms and is used in the Solaris operating system for all time-dependent scheduling decisions, as is typical of UNIX implementations. Timeouts and time-slicing cannot be performed at any finer resolution than 10 ms. As a result, service-times and deadlines cannot be accurately supported at a finer granularity than 10 ms in the current prototype implementation.

We measured the cost of various systems calls used in adjusting the scheduling parameters of a given task and found that their overheads are comparatively small relative to the resolution of the clock interrupt timer. These scheduling parameters include shares, deadlines, time-quanta, etc. The cost of reading the scheduling parameters of a given task is about 10 microseconds. The cost of assigning new scheduling parameters to a task is about 20 microseconds. Finally, the cost of actually scheduling a task using the SMART prototype depends on the particular mix of tasks that need to be scheduled, but is about 25-50 microseconds for typical desktop workstation loads.

The remainder of this section highlights some performance results for various mixes of real-time and conventional resource requests. These requests were generated using a set of simple applications that allow us to vary their resource requirements to demonstrate the effectiveness of SMART under a variety of workloads. We look at conventional tasks, real-time tasks, and a combination of both in Sections 7.1, 7.2, and 7.3 respectively. We demonstrate that proportional sharing is achieved for all the cases, regardless of whether the real-time requests present (if any) have overloaded the system. We show in both Sections 7.2 and 7.3 that the scheduler drops the minimum number of deadline requests to achieve fair sharing, in the case of overload. Finally, we also show in Section 7.3 that latency tolerance helps minimize the number of deadlines dropped.

7.1 Conventional Applications

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

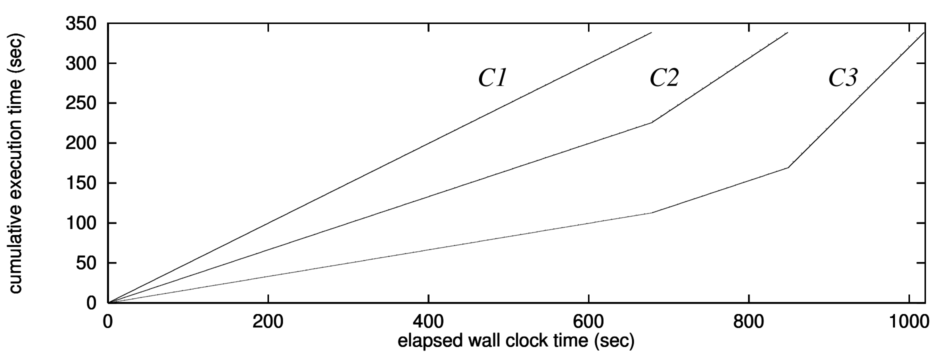


Figure 1 Proportional sharing for conventional applications

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

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

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

Task *C3*: $676 + 170 + (338 - (676 \times 1/6) - 170 \times 1/3) = 1014$ seconds.

The corresponding ideal performance curves are also plotted as dotted lines in Figure 1. However, the dotted ideal curves are hardly visible because of the close match with the actual experimental results. The results match the expected behavior very well, with tasks *C1*, *C2*, and *C3* finishing at times 678, 848, and 1018 seconds, respectively. Furthermore, the slopes of the graphs, indicating the resource consumption rates, show that the tasks are serviced in a proportionally fair manner throughout the execution. As expected, the slopes of *C1*, *C2*, and *C3* are of the ratio 3.00 : 2.00 : 1.00 in the first stage of the computation when all the programs are running. The slopes of *C2* and *C3* are of the ratio 2:00 : 1:00 in the second stage when tasks *C2* and *C3* are running.

7.2 Real-time Applications

Optimal Performance in Underload

To demonstrate the performance of SMART for real-time applications in underload, we executed two real-time applications, *R1* and *R2*, with periodic resource requests. The resource requests were event-driven, with the event arrival interval determining the deadline of the respective request. *R1* required 28-30 ms of execution time to complete each resource request, with each resource request having a 40 ms deadline from its instantiation. *R2* required 18-20 ms of execution time to complete each resource request, with each resource request having a 90 ms deadline from its instantiation. *R1* was given 2000 events to process and *R2* was given 888 events to process. *R1* and *R2* were both assigned equal shares, though the assignment of shares makes no difference in this case. We logged the number of missed deadlines and the cumulative execution time obtained by each task, which is illustrated by the solid line curves in Figure 2(a).

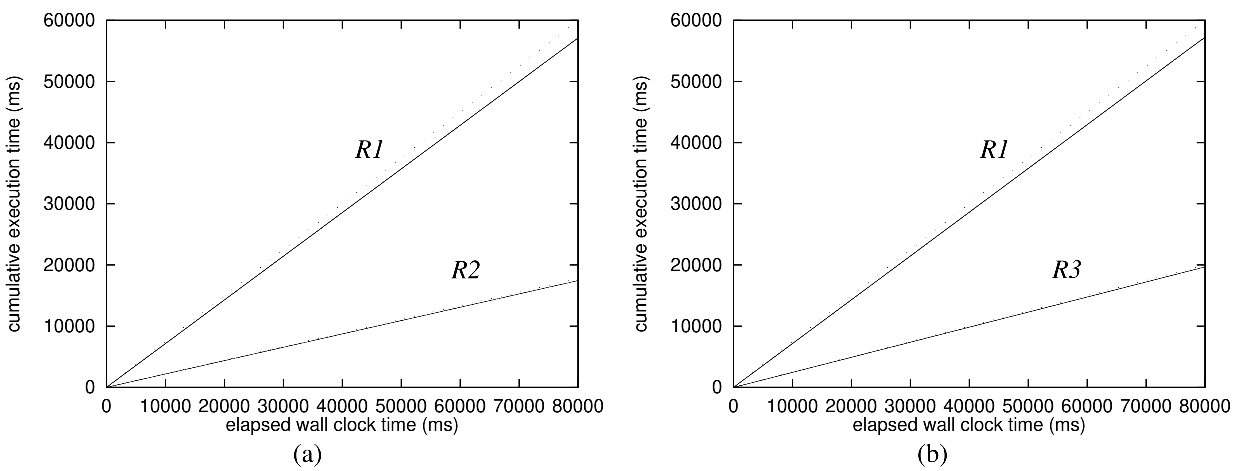


Figure 2 SMART scheduling of real-time applications in underload

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

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

Task *R2*: $888 \times 20 = 17760$ ms

The corresponding ideal performance curves are represented by the respective dotted lines in Figure 2(a). We can see in Figure 2(a) that both applications obtained nearly their ideal resource consumption rates, with the slight deviation from ideal being due to each task on average requiring slightly less than their respective ideal execution times to complete their resource requests. Task *R1* obtained 57,066 ms of execution time and task *R2* obtained 17,426 ms of execution time. More importantly, because SMART is optimal in underload, there were no missed deadlines for either application.

To demonstrate the optimal performance of SMART in underload, we executed two periodic real-time applications again, *R1* and *R3*, but this time the applications consume nearly 100% of the machine. In particular, *R3* requires 18-20 ms of execution time every 80 ms. *R1* was given 2000 events to process and *R2* was given 1000 events to process. *R1* and *R2* were both assigned equal shares, though the assignment of shares makes no difference in this case. We logged the number of missed deadlines and the cumulative execution time obtained by each task, which is illustrated by the solid line curves in Figure 2(b).

If the system were perfect and the actual processing time required in any given interval never exceeds 100% utilization, we would expect all deadlines to be satisfied in this case as well; both tasks should be able to obtain their requested resources. Furthermore, if *R1* required exactly 30 ms of execution time for each 40 ms deadline and *R2* required exactly 20 ms of execution time for each 80 ms deadline, we would expect both tasks to finish in roughly 800 seconds with the following cumulative execution times:

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

Task *R2*: $1000 \times 20 = 20000$ ms.

The corresponding ideal performance curves are represented by the respective dotted lines in Figure 2(b). We can see in Figure 2(b) that both applications obtained nearly their ideal resource consumption rates, with the slight deviation from ideal being due to each task on average requiring slightly less than their respective ideal execution times to complete their resource requests. Task *R1* obtained 57,178 ms of execution time and task *R2* obtained 19,673 ms of execution time. SMART met all of the deadlines of *R3* and only 3 out of 1999 of *R1*'s deadlines were missed. Even with nearly 100% average resource utilization and some variability in the execution times of these applications, SMART was able to meet over 99% of the deadlines.

Proportional Sharing in Overload

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

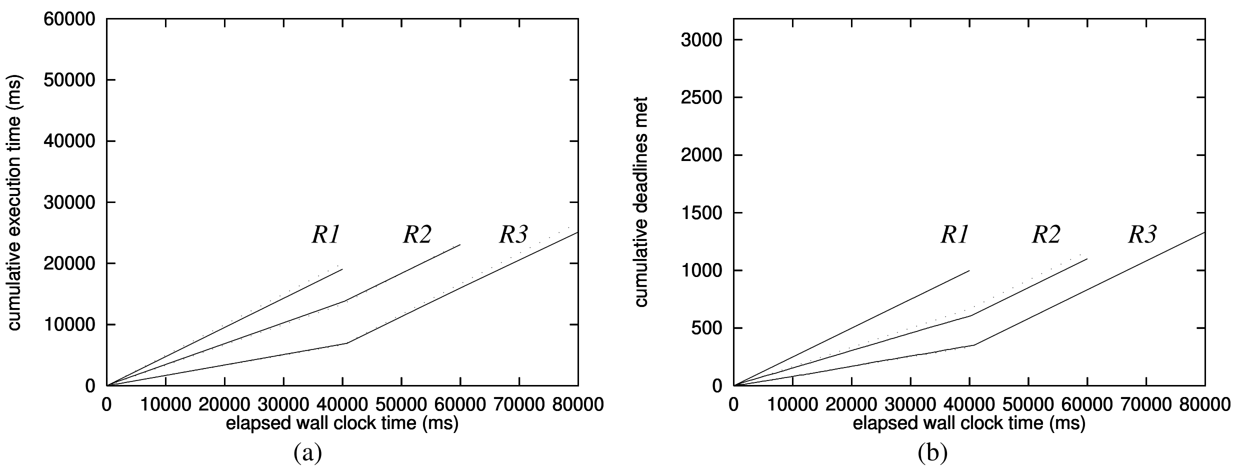


Figure 3 SMART proportional sharing among real-time applications in overload then underload

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

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

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

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

The corresponding ideal performance curves showing cumulative execution time for each task are plotted as dotted lines in Figure 3(a). As the system is overloaded during the first 40,000 ms, not all deadlines can be met. Based on their respective resource requirements and shares, we would expect the total number of deadlines met by each task to be:

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

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

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

The corresponding ideal performance curves showing the number of deadlines met for each task are plotted as dotted lines in Figure 3(b). These ideal numbers assume not only that each task receives the ideal amount of resources, but also that all of that processing time can be effectively used toward meeting deadlines; none of the processing time should be wasted on deadlines that will not be met. To do this, it is important to not only dole out processing time in the right sized quanta for each task, but also that the quanta be doled out at the right time to be synchronized with the deadlines.

We can see there is a close match between the measured results and the ideal curves. The total cumulative execution times of *R1*, *R2*, and *R3* were 19,045, 23,061, and 25,094 ms, respectively. The total number of deadlines met by *R1*, *R2*, and *R3* were 999, 1100, and 1331, respectively.

We divide the results into the overloaded and underloaded periods. During the first 40 seconds of elapsed wall clock time, all three applications are executing. The slopes of the left graph show the respective measured resource consumptions of *R1*, *R2*, and *R3* to be in the ratio 2.80 : 2.02 : 1.00. Note that the resource consumption of *R1* is a bit less than its proportional share because it does not require its proportional share to complete its deadlines. If the system were perfect and that there are no variability in the tasks' execution time, we expect applications *R1*, *R2*, and *R3* to miss roughly 0, 1/3, and 2/3 of the deadlines, respectively. During this period, each application had 1000 deadlines that it desired to have satisfied. Figure 3(b) shows the number of deadlines met. During the first 40 second period, *R1* met 999 deadline, *R2* met 604 deadlines, and *R3* met 345 deadlines. They correlate well with the ideal values.

R1 completes its execution at the beginning of the next 20 seconds of wall clock time while *R2* and *R3* continue to compete for resources. Note that the remaining tasks are no longer requesting more than their proportional shares. The slopes of Figure 3(a) show the respective measured resource consumptions of *R2* and *R3* during this period to be in the ratio 1.02 : 1.00. Both tasks miss a couple of deadlines as R1 is completing its execution, but then are able to satisfy all of their remaining deadlines. This example shows SMART automatically adjusts to the load condition and transitions between overload and underload gracefully.

7.3 Conventional and Real-time Applications

Real-time Requests Using Less than Proportional Share

In mixing conventional and real-time applications, we first consider the case when the real-time applications require less than their proportional share of resources to satisfy their deadlines. We show two examples of this case in Figure 4.

Figure 4(a) shows two equal-share applications, *R1* and *C1*, of which *R1* is real-time and *C1* is conventional. In particular, *R1* required 18-20 ms of execution time for each of its resource requests, and has a 40 ms deadline from its instantiation. If the system were perfect and *R1* required exactly 20 ms to process each 40 ms deadline request, we would expect *R1* not to miss any deadlines and for each application to use 50% of the system, as illustrated by the dotted line in Figure 4(a). Our measured results correspond well with the ideal and indicate that *R1* did not miss any of its deadlines. Since *R1* actually required slightly less than its proportional share of resources, *C1* was allowed to consume more than its proportional share and make better progress. In total, *R1* consumed roughly 48% of the system while *C1* consumed the remaining 52% of the system.

Figure 4(b) shows a periodic real-time application *R2* and a conventional application *C2* whose respective shares are in the ratio 3 : 1. *R2* required 28-30 ms of processing time with a periodic deadline every 40 ms. If the system were perfect and *R2* required exactly 30 ms to process each 40 ms deadline request, we would expect *R2* not to miss any deadlines and for *R2* and *C2* to consume resources in proportion to their shares, 75% and 25%, respectively, as illustrated by the dotted lines in Figure 4(b). Our measured results correspond well with the ideal and indicate that *R2* did not miss any of its deadlines. *R2* consumed roughly 72% of system while *C2* consumed the remaining 28% of the system. The slight difference from ideal is due to the fact that *R2* consumes a little bit less than the stated ideal.

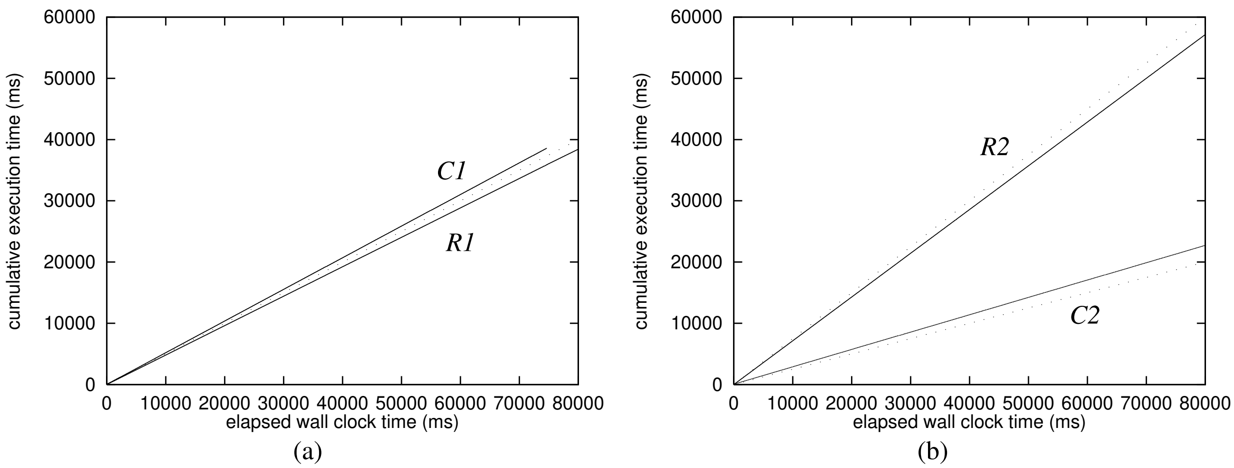


Figure 4 SMART proportional sharing among conventional and real-time applications

Latency Tolerance

To show some of the benefit of latency tolerance, we again consider the case of two equal-share applications, *R1* and *C1*, of which *R1* is real-time and *C1* is conventional. In this case however, while each resource request of *R1* still has a 40 ms deadline from its instantiation, the work required for each resource request varies with an even distribution between 10-30 ms of execution time, with an average of 20 ms. While the resource consumption graph in this case remains similar to Figure 4(a) given the time scales involved, the number of deadlines that are missed by *R1* depends on the latency tolerance of *C1*. If *C1* cannot tolerate any latency, then *R1* misses 192 out of 1999 deadlines. However, if *C1* is given a latency tolerance of just 100 ms, *R1* can execute without missing any of its deadlines, despite the fact that its desired resource consumption for any given resource request varies from 25% to 75% of the machine.

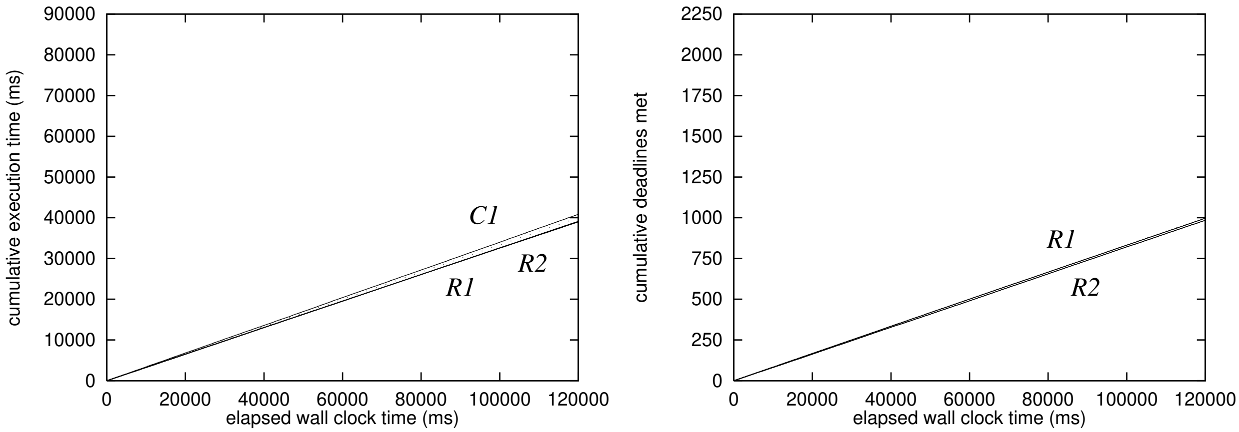
Proportional Sharing with Latency Tolerance in Overload

We demonstrate that SMART is able to share resources proportionally even for mixes of conventional and real-time applications in which the set of real-time resource requests is overloaded. Figure 5 shows the results of executing three equal-share applications, *R1*, *R2*, and *C1,* of which *R1* and *R2* are real-time and *C1* is conventional. In particular, *R1* and *R2* are identical applications, each requiring roughly 38-41 ms of execution time for each resource request, and each resource request having a 60 ms deadline from its instantiation. Each real-time task processed a sequence of 2000 resource requests. *C1* is a conventional task with a latency tolerance of 200 ms and takes 40,000 ms of processing time to complete.

If the system were perfect, we expect tasks to share resources based on their shares since the system is overloaded. As a result, each task should be allocated 1/3 of the resources. All three tasks should complete in 120,000 ms. Their ideal performance curves fall on the dotted line in Figure 5(a). Given 1/3 of the resources, each real-time task should be able to complete half of its deadlines:

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

Figure 5(a) shows that all three applications consume nearly identical proportions of the resource, in close correspondence with the ideal. *C1* was able to consume a slightly larger portion of the processing time because it was always able to run if there were available resources. *R1* and *R2* were only able to make valuable use of the processing time when it was possible to satisfy their respective deadlines. For instance, if *R1* still had to run 40 ms to meet its deadline and the deadline was only 30 ms away, it would be useless to run *R1*.



(a) (b)

Figure 5 SMART proportional sharing for conventional and real-time applications in overload

Figure 5(b) shows that the number of deadlines met by *R1* and *R2* correspond well with their expected values. Out of 2000 deadlines, *R1* met 985 deadlines while *R2* met 998 deadlines. The number of respective deadlines that each of them completed is in the ratio 1.00 : 1.01, which very closely corresponds to their shares. Over the course of 120 seconds of execution, the total amount of time spent by both *R1* and *R2* in processing deadlines that could not be met was less than 1 second. The SMART scheduler is able to make efficient use of processor cycles to devote the processor allocations of both real-time applications almost entirely toward resource requests whose deadlines can be met.

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

7.4 Summary

We have provided measured experimental results of the SMART prototype implemented in the Solaris operating system that demonstrate its effectiveness for both real-time and conventional applications. The results show that the scheduler can in practice provide nearly optimal performance for real-time applications in underload, and provides accurate proportional sharing across both real-time and conventional applications, and even allows real-time applications to satisfy their deadlines in a proportional manner when not all deadlines can be met. Good results are obtained despite the fact that the current prototype scheduler is unoptimized. This shows that this high degree of flexible control and efficient resource utilization can be achieved in practice without prohibitive scheduling overhead.

8 Concluding Remarks

In this paper, we have described the design of SMART, a processor scheduler for supporting both real-time and conventional resource requirements, such as those found in multimedia applications. Unlike previous approaches to supporting multimedia applications, the design combines the desirable properties of proportional sharing, latency tolerance, and best-effort real-time scheduling in an integrated manner.

We have implemented a SMART prototype in the Solaris operating system. Our measured performance results of the implementation demonstrate its effectiveness in a real system to make efficient use of hardware resources and its unique ability to provide proportional sharing across both real-time and conventional applications.

9 Acknowledgments

We thank James G. Hanko, J. Duane Northcutt, and Gerald A. Wall for many helpful discussions. We also thank Jim for providing much appreciated Solaris kernel expertise. This work was supported in part by an NSF Young Investigator Award and Sun Microsystems Laboratories. Solaris is a registered trademark of Sun Microsystems, Inc. SPARCstation and hyperSPARC are trademarks of SPARC International, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

References

1. AT&T: *UNIX System V Release 4 Internals Student Guide, Vol. I, Unit 2.4.2.*, AT&T, 1990.
2. R. K. Clark: *Scheduling Dependent Real-Time Activities*, Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, August 1990.
3. A. Demers, S. Keshav, S. Shenker: *Analysis and Simulation of a Fair Queueing Algorithm*, Proceedings of SIGCOMM '89, September 1989.
4. 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, November 1994.
5. J. G. Hanko, E. M. Kuerner, J. D. Northcutt, G. A. Wall: *Workstation Support for Time-Critical Applications*, Proceedings of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video, November 1991.
6. J. G. Hanko: *A New Framework for Processor Scheduling in UNIX*, Abstract talk from the Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video, November 1993.
7. G. J. Henry: *The Fair Share Scheduler*, Bell Laboratories Technical Journal, October 1984.
8. K. Jeffay, D. Bennett: *A Rate-Based Execution Abstraction for Multimedia Computing*, Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video, April 1995.
9. M. B. Jones, P. J. Leach, R. P. Draves, J. S. Barrera, III: *Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System*, Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video, April 1995.
10. L. Kleinrock: *Queueing Systems Vol 2: Computer Applications*, John Wiley & Sons, Inc., 1976.
11. 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, 1989.
12. J. P. Lehoczky, L. Sha, J. K. Strosnider: *Enhanced Aperiodic Responsiveness in Hard Real-Time Environments*, Proceedings of the IEEE Real-Time Systems Symposium, December 1987.
13. R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf: *Policy/Mechanism Separation in Hydra*, Proceedings of the Fifth Symposium on Operating Systems Principles, ACM, November 1975.
14. C. L. Liu, J. W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, JACM 20(1), January 1973.
15. C. D. Locke: *Best-Effort Decision Making for Real-Time Scheduling*, Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, May 1986.
16. 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, May 1994.
17. 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, November 1993.
18. J. Nieh, M. S. Lam: *Integrated Processor Scheduling for Multimedia*, Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video, April 1995.
19. J. D. Northcutt: *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*, Academic Press, Boston, 1987.
20. 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, June 1993.
21. S. Ramos-Thuel, J. P. Lehoczky, *On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems*, Proceedings of the IEEE Real-Time Systems Symposium, December 1993.
22. L. Sha, J. P. Lehoczky, R. Rajkumar: *Solutions for Some Practical Problems in Prioritized Preemptive Scheduling*, Proceedings of the IEEE Real-Time Systems Symposium, December 1986.

23. I. Stoica, H. Abdel-Wahab: *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*, Technical Report 95-22, Department of Computer Science, Old Dominion University, November 1995.
24. C. A. Waldspurger, W. E. Weihl: *Lottery Scheduling: Flexible Proportional-Share Resource Management*, Proceedings of the First Symposium on Operating Systems Design and Implementation, November 1994.
25. C. A. Waldspurger: *Stride Scheduling: Deterministic Proportional-Share Resource Management*, Technical Memorandum MIT/LCS/TM-528, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1995.
26. G. A. Wall, J. G. Hanko, J. D. Northcutt: *Bus Bandwidth Management in a High Resolution Video Workstation*, Proceedings of the Third International Workshop on Network and Operating Systems Support for Digital Audio and Video, November 1992.