

SMART UNIX SVR4 Support for Multimedia Applications

Jason Nieh^{1,2} and Monica S. Lam¹

¹Computer Systems Laboratory, Stanford University

²Sun Microsystems Laboratories

Abstract

Multimedia applications have dynamic and adaptive real-time requirements. Current scheduling practice, as typified by UNIX System V Release 4, lacks the necessary information and interfaces to meet these requirements. To address this problem, we have created the SMART (Scheduler for Multimedia And Real-Time) interface. It explicitly accounts for application-specific time constraints and provides dynamic feedback from the scheduler to applications to allow them to adapt to the system loading condition. This paper describes the design of the interface and its implementation in the Solaris UNIX operating system to provide an effective SVR4-conformant full featured environment for supporting multimedia applications.

1 Introduction

Multimedia applications depend on operating systems to manage resources to support their dynamic and adaptive real-time requirements. Anticipating that processor scheduling based on traditional time-sharing would not be suitable for addressing these requirements, commercial operating systems typified by UNIX System V Release 4 (SVR4) [9] provide a real-time static priority scheduler to support real-time applications in addition to a standard UNIX time-sharing scheduler for other applications. In UNIX SVR4, applications managed by the real-time scheduler are given higher priority than all other activities in the system, thereby allowing them to monopolize resources as needed to obtain fast response time.

Despite the presence of a so-called real-time scheduler, UNIX SVR4 scheduling has been experimentally demonstrated to be unacceptable for supporting multimedia applications [5]. Instead, it allows runaway real-time applications to cause basic system services to lock up, and the user to lose control over the machine. It provides little in the way of higher-level programming abstractions to deal with timing requirements, imposing a recurring cost

on each programmer who creates a real-time application. Because UNIX SVR4 serves as the most common basis for workstation operating systems, it is important to address its limitations, especially as multimedia applications increasingly populate the workstation desktop.

To reduce the burden of real-time programming and provide effective performance for multimedia applications, we have created SMART, a Scheduler for Multimedia And Real-Time. SMART provides a simple interface for applications and users that allows access to its underlying resource management mechanisms [6]. This interface (1) enables the operating system to manage resources more effectively by using knowledge of application-specific timing requirements, (2) provides dynamic feedback to real-time applications to inform them if their time constraints cannot be met so that they can adapt to the current loading condition, (3) gives end users simple predictable controls that can be used to bias the allocation of resources according to their preferences. We have implemented SMART in the Solaris UNIX SVR4 operating system, bringing effective support for multimedia applications to an SVR4-conformant, full-featured operating environment.

This paper focuses on the design of the SMART interface, how the interface supports the programming of real-time multimedia applications, and its resulting performance. We note that SMART also provides an effective scheduling algorithm that takes advantage of the SMART interface to support coexisting real-time and non-real-time computations. The scheduling algorithm is described in [6].

This paper is organized as follows. Section 2 provides a more detailed discussion of the requirements of multimedia applications. Section 3 presents the design of the SMART interface and its usage model. Section 4 describes the SMART real-time application programming interface in greater detail. Section 5 describes the implementation of the SMART interface in the Solaris UNIX SVR4 operating system. Section 6 presents some experimental results that demonstrate the improved performance achievable using the SMART interface over standard UNIX SVR4

time-sharing and real-time scheduling. Section 7 describes related work. Finally, we present some concluding remarks.

2 Multimedia applications

To understand the requirements imposed by multimedia applications on the design of an operating system interface, we discuss the dynamic and adaptive real-time nature of these applications. We especially focus on the characteristics of applications that manipulate digitized audio and video data.

2.1 Timing characteristics

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

Not only are there timing requirements within a continuous media stream, but there may also be timing requirements among multiple media streams as well. These timing requirements are due to the need to synchronize multiple media streams. For instance, in playing a movie, the audio stream and the video stream need to be synchronized so that the desired audio is heard when a given video frame is displayed.

When an application processes and displays continuous media streams, it must typically meet two kinds of timing requirements to preserve the temporal alignment of the media streams being processed. One requirement is that any delay due to processing between the input media stream and the processed output media stream should be as constant as possible. Variance in the delay introduces undesirable jitter in the output stream. The other requirement is that the application must process media samples fast enough. If media samples are not processed at the rate at which they arrive, then they will be late being displayed and it will not be possible to maintain the exact temporal alignment of the media samples.

2.2 Dynamic characteristics

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

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

2.3 Adaptive characteristics

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

3 SMART usage model and interface

The SMART interface provides two kinds of support for multimedia applications. One is to support the developers of multimedia applications that are faced with writing applications that have dynamic and adaptive real-time

requirements. The other is to support the end users of multimedia applications, each of whom may have different preferences for how a given mix of applications should run.

3.1 Application developer support

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

To address these limitations, SMART provides to the application developer three kinds of programming constructs: a *time constraint* to allow an application to express to the scheduler the timing requirements of a given block of application code, a *notification* to allow the scheduler to inform the application via an upcall when its timing requirements cannot be met, and an *availability* to indicate the availability of processing time. In particular, applications can have blocks of code that have time constraints and blocks of code that do not, thereby allowing application developers to freely mix real-time and non-real-time computations. The SMART application programming constructs are described in further detail in Section 4.

By allowing applications to inform the scheduler of their time constraints, the scheduler can optimize how it sequences the resource requests of different applications to meet as many time constraints as possible. It can delay those computations with less stringent timing requirements to allow those with more stringent requirements to execute. It can use this knowledge of the timing requirements of all applications to estimate the load on the system and determine which time constraints can and cannot

be met. By providing notifications, the scheduler frees applications from the burden of second guessing the system to determine if their time constraints can be met. By having the scheduler provide information on the availability of resources to applications, an adaptive real-time application can determine how best to adjust its execution rate when its timing requirements cannot be met.

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

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

3.2 End user support

Different users may have different preferences for how processing time should be allocated among a set of applications. Not all applications are always of equal importance to a user. For example, a user may want to ensure that an important video teleconference be played at the highest image and sound quality possible, at the sacrifice if need be of the quality of a television program that the user was just watching to pass the time. However, current practice, as typified by UNIX, provides little in the way of predictable controls to bias the allocation of resources in accordance with user preferences. For instance, in UNIX time-sharing, all that a user is given is a “nice” knob [9] whose setting is poorly correlated to user observable behavior [5].

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

The user can specify that applications have different priorities, meaning that the application with the higher priority is favored whenever there is contention for resources. The system will not degrade the performance of a higher priority application to execute a lower priority application. For instance, suppose we have two real-time applications, one with higher priority than the other, and the lower pri-

ority application having a computation with a more stringent time constraint. If the lower priority application needs to execute first in order to meet its time constraint, the system will allow it to do so as long as its execution does not cause the higher priority application to miss its time constraint. Among applications at the same priority, the user can specify the share of each application, resulting in each application receive an allocation of resources in proportion to its respective share whenever there is contention for resources.

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

3.3 Summary

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

While SMART accounts for both application and user information in managing resources, it in no way imposes draconian demands on either application developers or end users for information they cannot or choose not to provide. The design provides reasonable default behavior as well as incrementally better results for incrementally more information. By default, an end user can just run an application as he would today and obtain fair behavior. If he desires that more resources should be allocated to a given application, SMART provides simple controls that can be used to express that to the scheduler. Similarly, an application developer need not use any of SMART's real-time programming constructs unless he desires such functionality. Alternatively, he might choose to use only time

constraints, in which case he need not know about notifications or availabilities. When the functionality is not needed, the information need not be provided. However, unlike other systems, when the real-time programming support is desired, as is often the case with multimedia applications, SMART has the ability to provide it.

4 SMART real-time API

Having described the basic usage model for SMART and presented an overview of the SMART interface, we now provide a more detailed description of the real-time application programming constructs and their use. An example that shows how these constructs are used in a real-time video application is described in Section 6.8.

4.1 Time constraint

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

- *deadline*: The deadline is the time by which the application requests that the block of code be completed.
- *cpu-estimate*: The cpu-estimate is an estimate of the amount of processing time required for the block of code.

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

4.2 Notification

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

- *notify-time*: The notify-time is the time after which the scheduler should inform the respective application if it is unlikely to complete its computation before its deadline.
- *notify-handler*: The notify-handler is a function that the application registers with the scheduler. It is invoked via an upcall mechanism from the scheduler when the scheduler notifies the application that its deadline cannot be met.

The notify-time is used by the application to control when the notification upcall is delivered. For instance, if the notify-time is set equal to zero, then the application will be notified immediately if early estimates by the

scheduler indicate that its deadline will not be met. On the other hand, if the notify-time is set equal to the deadline, then the application will not be notified until after the deadline has passed if its deadline was not met.

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

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

4.3 Availability

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

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

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

cessing time and thereby deliver a higher quality of service.

5 Implementation

We have implemented the SMART interface in Solaris 2.5.1, the current release of Sun Microsystems's UNIX SVR4 operating system. The interface implementation is based upon two UNIX SVR4 system calls, `prIOCtl` and `signal`. Our implementation did not require the creation of any new system calls and is fully SVR4-compliant.

The UNIX SVR4 `prIOCtl` system call allows an arbitrary set of parameters to be passed between applications and users and the scheduler. In this way, the UNIX SVR4 scheduling interface is extensible, allowing new scheduling parameters to be used when new underlying scheduling mechanisms are added to support them. In particular, we use the `prIOCtl` system call to implement all aspects of the SMART interface except notifications. We modified the internal UNIX SVR4 scheduler to provide the necessary support for the SMART interface while maintaining support for the scheduling parameters used by the standard UNIX SVR4 scheduler.

Notifications are implemented using both `prIOCtl` and `signal` system calls. `prIOCtl` is used to set the notify-time of a notification while the `signal` interface is used to register the notify-handler and send the notification upcall when required. Writing a notify-handler to process a notification upcall is the same process as writing a generic signal handler.

An important modification to the UNIX SVR4 internals that was necessary to effectively support the SMART interface was to provide a higher resolution timer mechanism. The standard UNIX SVR4 scheduling framework upon which the Solaris operating system is based employs a periodic 10 ms clock tick. Without any modification, this mechanism would limit the granularity of time constraints and other time-dependent programming constructs to 10 ms. Such a limitation is problematic for supporting multimedia applications whose timing requirements often require a finer granularity. To address this problem, we added a high resolution timeout mechanism to the kernel and reduced the time scale at which timer based interrupts can occur. The exact resolution allowed is hardware dependent. For example, on a SPARCstation 10 workstation, which is used in the experiments described in Section 6, the resolution is 1 ms.

We measured the cost of assigning scheduling parameters such as time constraints or reading scheduling information such as availabilities. It is small. The exact overhead is hardware dependent, but for example, using a system with only a 150 MHz hyperSPARC processor, the cost of assigning scheduling parameters to a process is

20 μ s while the cost of reading the scheduling information for a process is only 10 μ s.

6 Experimental results

We present some experimental data to demonstrate how the performance of multimedia applications can be improved using the SMART interface and compare these results with those achievable with the standard UNIX SVR4 time-sharing (TS) and real-time (RT) schedulers. As a representative multimedia application, we used the SLIC-Video player, a low-cost video product that captures and displays video images in real-time. As a baseline, we measure the performance of the application when running on an otherwise quiescent system. We then measure the performance of multiple SLIC-Video players running under UNIX TS and UNIX RT with a dynamically changing load. To improve the video performance, we describe a few simple modifications to the application that allow it to take advantage of the SMART interface, then present results to quantify the performance improvement achieved.

6.1 Application description and quality metric

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

The digitizer board has a limited amount of buffering that allows the hardware to continue to process an analog video stream into video frames while the software captures video data from the digitizer board. The buffer has three slots that are organized as a ring; when it is full, the hardware wraps around to the begin of the buffer and overwrites its contents. Each slot is assigned a timestamp when it is written. Locking is used to ensure that the hardware does not overwrite a buffer slot that is being read by the software and the software does not read a buffer slot that is in the middle of being written by the hardware. In

normal playback mode, the hardware digitizer cooperates with the software capture by sending a signal each time the hardware completes digitizing a frame. Upon receiving the signal, the software follows a policy of reading from the buffer slot with the earliest timestamp. For our experiments, the arrival rate of frames from the hardware to the software is 29.97 frames per second (fps).

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

6.2 Experimental testbed

The experiments were performed on a standard, production SPARCstation 10 workstation with a single 150 MHz hyperSPARC processor, 64 MB of primary memory, and 3 GB of local disk space. Three SLIC-Video capture cards were added to the system to permit the execution of three video players showing three different video sources at the same time. The video sources used were a video camera and television programming from a Sun Tuner. 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 current release of Sun's operating system, Solaris 2.5.1, was used as a basis for our experimental work. The high resolution timing functionality described in Section 5 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 lightweight mechanism that writes timestamped event identifiers into a memory log. The timestamps are at 1 μ s resolution. We measured the cost of the mechanism on the testbed workstation to be 2-4 μ s 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 was restarted prior to each experimental run. Each experimental run lasted 300 seconds.

6.3 Baseline performance

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

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

Table 1 **Baseline application performance**

The measurements show that SLIC-Video uses up nearly 90% of the CPU to display video at 29.97 320x240 pixel fps. It displays over 99% of the frames that arrive, and it does so in a timely manner that preserves the temporal alignment in the video stream. Both the delay between frame arrival and frame display, and the time delta between frames displayed have minimal variation. The delay between frame arrival and frame display is roughly two frame times, corresponding to the application policy of processing the frame in the digitizer hardware

buffer with the earliest timestamp. The time delta between displayed frames is one frame time, corresponding to the fact that over 99% of the frames that arrive are displayed.

6.4 Experimental scenario and ideal performance

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

Using the baseline performance measurements, we first describe what the expected ideal performance should be for this scenario. Since a single SLIC-Video player consumes nearly the entire machine, it is not possible to execute two video players at 30 fps. Instead, as it is best to maintain a more constant time delta between displayed frames, we would ideally expect that during the first 100 seconds of execution, V1 and V2 would each reduce their frame rate by skipping half of their respective frames and displaying the other half. Note that this does not require consuming 100% of the CPU. Ideally, the delay between frame arrival and display should be 100.10 ms and the time delta between displayed frames should be 66.73 ms. When V3 begins, we would expect V1 and V2 to reduce their frame rate further, displaying only 25% of their frames, with delay and time delta of 166.83 ms and 133.47 ms, respectively. V3 should be able to display 50% of its frames, with delay and time delta of 100.10 ms and 66.73 ms, respectively. Upon termination of V3, we would ideally expect that the performance of V1 and V2 would be the same as during the first 100 seconds of their execution. In all cases, there should ideally be zero variation in the time delay between frame arrival and display and the time delta between frame displays. The ideal CPU allocations and application results are shown in Figures 1 and 2, respectively.

6.5 UNIX SVR4 time-sharing performance

We ran the experimental scenario described in Section 6.4 under standard UNIX TS scheduling. To give V3 roughly twice the performance of V1 and V2 under UNIX TS, extensive trial and error was required to find a suitable “nice” setting for V3 to bias the allocation of resources in accordance with the proportions desired. The nice setting

for V3 was +15. The CPU allocations and application measurements for this experiment are shown in Figures 1 and 2, respectively.

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

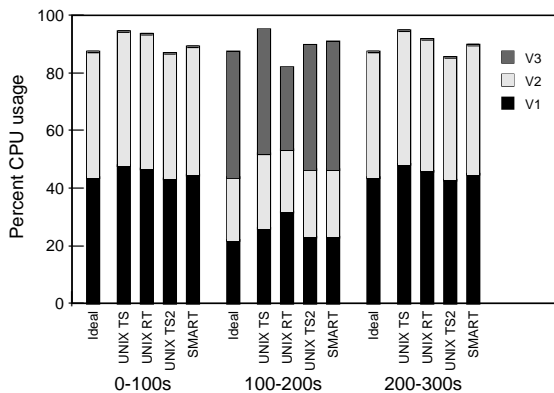


Figure 1 CPU allocation

6.6 UNIX SVR4 real-time performance

We ran the same experimental scenario of three video players under standard UNIX RT scheduling. V1 and V2 are both assigned the same default priority, while V3 is assigned a higher priority than either V1 or V2. The CPU allocations and application measurements for this experiment are contrasted with the results under UNIX TS in Figures 1 and 2, respectively.

While the standard deviations in the quality metrics are better than those under UNIX TS, UNIX RT suffers from two major problems. The first problem is evident by the performance measurements on V3. While its performance should be twice that of V1 or V2, in fact its performance is actually somewhat worse than either of the other video players. This is in spite of the fact that V3 has a higher priority than either V1 or V2. The problem is that the sig-

naling mechanism used to inform the user-level application of the arrival of a frame is a system-level function that therefore executes at a system-level priority. In UNIX SVR4, processes that are scheduled by the real-time scheduler are given higher priority than even system functions. The result is that the signal mechanism does not get to execute until all of the real-time video players finish processing their respective frames and block waiting to be informed of the arrival of a new frame to be processed. This serializes the execution of all of the video applications, irrespective of their assigned priority. The result for the end user is a complete lack of control in biasing the allocation of resources according to his preferences.

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

6.7 Managing time in UNIX SVR4 time-sharing

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

In an attempt to address these problems and improve the performance under UNIX TS, the video application was modified to account for the frame arrival times in determining which frames it should render and when it should render each of those frames. The application selects a time delay in which to render its video frames. It measures the amount of wall clock time that elapses during the processing of each frame. Then, it uses an exponential average of the elapsed wall clock time of previously displayed frames as an estimate of how long it will take to process the current frame. This estimate is used in conjunction with the frame arrival time to determine if the given frame can be displayed on time. If the

video player is ready to display its frame early, then it delays until the appropriate time; but if it is late, it discards its current frame. The application defines early and late as more than 16.68 ms (half of the time delta between arriving frames) early or late with respect to the selected time delay.

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

We ran the same experimental scenario of three video players under standard UNIX TS with the above mentioned code modifications. The results are shown in Figures 1 and 2 as “UNIX TS2”. We see that the performance is better than UNIX TS without explicit time management in the application, and does not have the pathological behavior found with UNIX RT. However, the standard deviation in time delay for V1 and V2 while V3 is running is still more than 40 ms, which is far beyond the modest 16.83 ms threshold of timeliness used by the application. This is the result of two problems. One is that the scheduler, having no knowledge of the timing requirements of the application, does not allocate resources to each application at the right time. The other problem is that the application has a difficult time of selecting the best time delay and frame rate to use for the given loading condition. Without scheduler information, it must guess at when the load changes based on its own estimates of system load. The result is that its selected time delay and frame rate oscillate back and forth due to inaccurate knowledge of the allocation of processing time that the scheduler will give the application under the given system load. Guessing is not good enough.

6.8 Programming with the SMART real-time API

To enable the SLIC-Video application to take advantage of SMART, three simple modifications were made to the code described in Section 6.7. First, rather than having the application rely on its own estimates of whether or not a frame is late and should be discarded, the application sets a time constraint that informs the scheduler of its deadline and cpu-estimate. The deadline is set to be 16.68

ms after the selected time delay. The cpu-estimate is calculated in the same manner as the average elapsed wall clock time: the application measures the execution time required for each frame and then uses an exponential average of the execution times of previously displayed frames as the cpu-estimate.

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

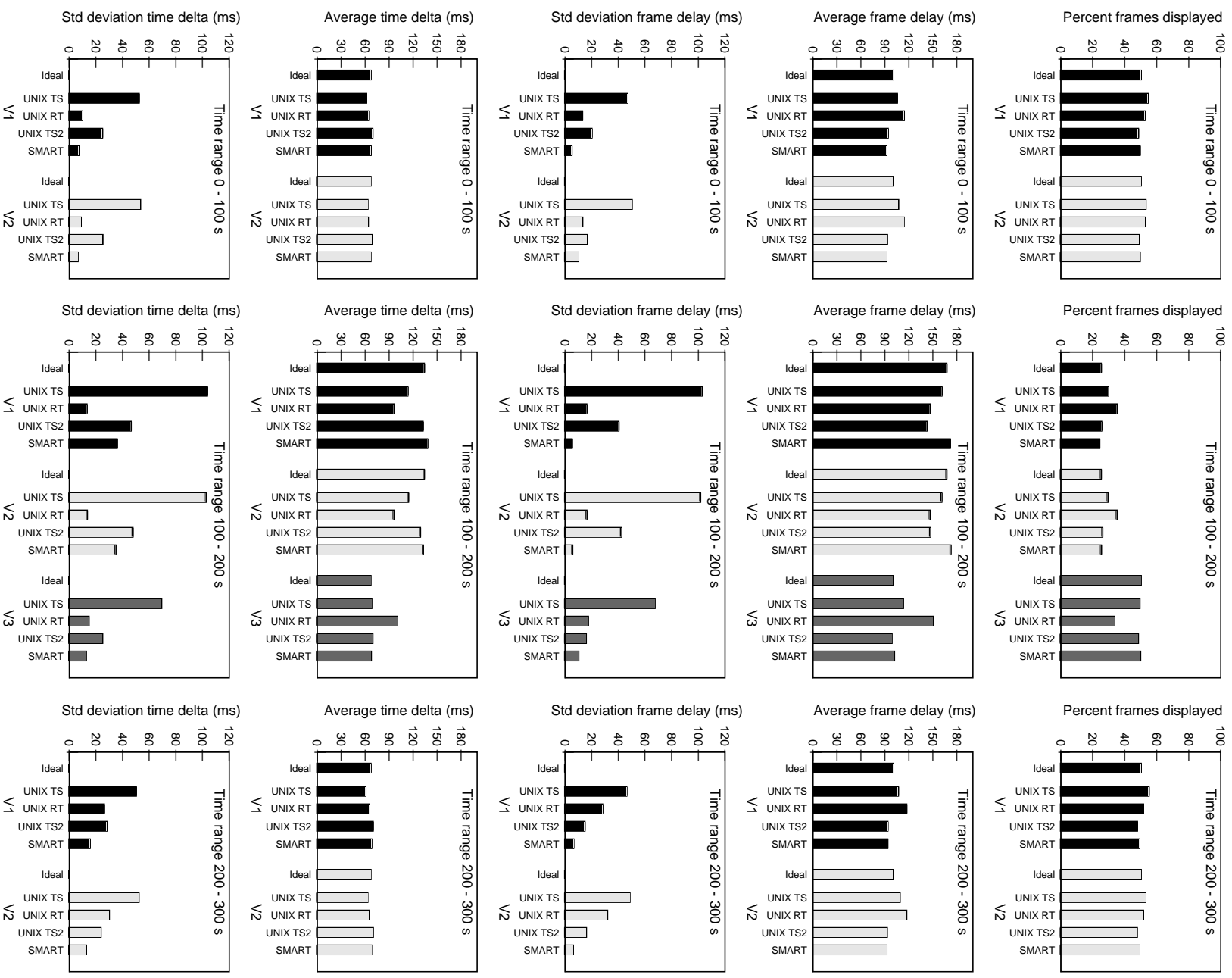
Third, rather than having to guess what the system loading condition is at any given moment, the application obtains its availability from the scheduler. It reduces its frame rate if frames cannot be completed on time and the required computational rate to process frames on time at the current frame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

6.9 SMART UNIX SVR4 performance

We ran the same experimental scenario of three video players under SMART, taking advantage of its real-time API. The CPU allocation and application results of this experiment are shown in Figures 1 and 2, respectively. Not only does SMART effectively allocate CPU time in accordance with the user preferences for the experiment, SMART provides application results that are closest to the ideal performance figures.

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

Figure 2 Application performance



7 Related work

Many approaches have been tried for supporting the processing requirements of multimedia applications. A commonly proposed resource management abstraction for these applications is resource reservations [4][8]. Reservations are used to allow each application to request a percentage of the CPU. Real-time requirements are typically assumed to be periodic, and the reservation percentage is set equal to the ratio of the processing time required and its associated deadline. They are combined with admission control to ensure that the total reservations sum up to no more than 100%. The problem is that reservations are an overly static abstraction for the dynamic requirements of multimedia. They simply deny access to later arriving applications, providing insufficient feedback to allow real-time application to adapt to the system loading condition. If users have different preferences, this information must be collapsed with all of the timing information into a single reservation value. As a result, significant information is lost, denying the scheduler the knowledge it needs to manage resources effectively.

Rialto [1] attempts to combine reservations with time constraints to ensure that the scheduler has access to application-specific timing requirements. Unlike SMART, its support for adaptive real-time applications is limited as it provides no asynchronous notification mechanism to inform applications if their time constraints cannot be met. To overcome the static limitations of reservations, it proposes renegotiating the reservations over time, but provides no policy for doing so to meet the dynamic requirements of multimedia applications.

Recognizing the limitations of a reservation model, the VuSystem [3] provides a framework that allows applications to cooperatively and dynamically adjust their execution rates. However, the system is only an application-level framework. It relies on the underlying operating system to manage resources in a timely manner, but does not give the operating system the necessary timing information to do that job effectively. Because SMART propagates timing information down to the scheduler where it is needed, it can deliver superior performance.

8 Conclusions

We have described a new scheduling facility that provides effective support for multimedia applications in an SVR4-conformant, full-featured environment. It is implemented in the Solaris UNIX operating system. Our measurements of actual application performance demonstrate its improved effectiveness over standard UNIX SVR4 scheduling for multimedia applications. The effectiveness of the solution is the result of a rich interface

that accounts for user preferences and allows applications to cooperate with the scheduler in supporting their real-time requirements. This cooperation provides the scheduler with the application-specific timing information it needs to manage resources effectively, and provides real-time applications with the necessary dynamic feedback to enable them to adapt to changes in the system load to provide the best possible quality of service.

Acknowledgments

We thank James G. Hanko, J. Duane Northcutt, and Gerard A. Wall for many helpful discussions. Much of the scheduler interface arose out of those discussions. This work was supported in part by an NSF Young Investigator Award and Sun Microsystems Laboratories.

References

1. M. B. Jones, J. S. Barrera III, A. Forin, P. J. Leach, D. Rosu, M. C. Rosu, "An Overview of the Rialto Real-Time Architecture", *Proceedings of the Seventh ACM SIGOPS European Workshop*, Sept. 1996.
2. K. B. Kenny, K. Lin, "Building Flexible Real-Time Systems Using the Flex Language", *IEEE Computer*, May 1991.
3. C. J. Lindblad, "A Programming System for the Dynamic Manipulation of Temporally Sensitive Data", Technical Report MIT/LCS/TR-637, Laboratory for Computer Science, Massachusetts Institute of Technology, Aug. 1994.
4. 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.
5. 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*, Nov. 1993.
6. J. Nieh, M. S. Lam, "The Design, Implementation, and Evaluation of SMART: A Scheduler for Multimedia Applications", Technical Report CSL-TR-97-721, Computer Systems Laboratory, Stanford University, Apr. 1997.
7. SLIC-Video User's Guide, Rel. 1.0, MultiMedia Access Corporation, 1995.
8. I. Stoica, H. Abdel-Wahab, K. Jeffay, "On the Duality between Resource Reservation and Proportional-Share Resource Allocation", *Proceedings of Multimedia Computing and Networking*, Feb. 1997.
9. UNIX System V Release 4 Internals Student Guide, Vol. I, Unit 2.4.2., AT&T, 1990.