Adaptive Rate-Controlled Scheduling for Multimedia Applications*

David K.Y. Yau and Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188
{yau,lam}@cs.utexas.edu

ABSTRACT

We present a framework for integrated scheduling of continuous media (CM) and other applications. The framework consists of a rate-controlled on-line CPU scheduler, an admission control interface, a monitoring module and a rate adaptation interface. Rate-controlled scheduling allows processes to reserve CPU time to achieve progress guarantees. It provides firewall protection between processes such that the progress guarantee to a process is independent of how *other* processes actually make scheduling requests. Rate adaptation allows a CM application to adapt its rate to changes in its execution environment. We have implemented the scheduling framework as an extension to Solaris 2.3. We present experimental results which show that our framework is highly effective in scheduling CM and various other applications in a general purpose workstation environment.

KEYWORDS: Continuous media, CPU scheduling, adaptive rate control, rate reservation, QoS guarantee, firewall property

1 INTRODUCTION

Advances in digital and networking technologies have enabled the integration of "continuous" media (CM) data, such as video and audio, with traditional "discrete" data types, such as graphics and text, in packet switching networks and general purpose workstations. System support for CM applications has recently received much attention [3, 7, 10].

CM applications require certain real-time constraints. They may interface with a media device (such as an audio codec or a video capture board) or with a network that transports media packets. Therefore, they need to process external events such as device interrupts or network interrupts in a timely manner.

Research supported in part by National Science Foundation under grant no. NCR-9506048, an equipment grant from the AT&T Foundation, and an IBM graduate fellowship awarded to David Yau for the 1996-97 academic year.

ACM Multimedia 96, Boston MA USA
© 1996 ACM 0-89791-871-1/96/11 ..$3.50

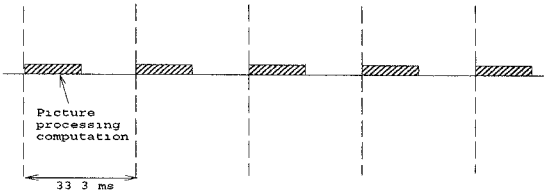

Figure 1: Execution profile of a periodic video application.

As an example, consider a video application that sends pictures to a network at a rate of 30 per second. A video capture board is connected to the computer on which the video application runs. Every 33.3 ms, the video capture board digitizes and compresses a picture, and buffers the compressed picture in on-board memory for reading by the video application. The video application reads the picture from the video capture board, packetizes the data and sends packets to the network. The execution profile of the application is shown in Fig. 1. The vertical lines mark the times at which new pictures are produced by the video capture board. The computation required by the video application to process each picture (which includes reading, packetizing and sending the picture data) is shown as a shaded box. For minimal delay and buffering inside the video capture board, processing of a picture should complete before the next picture is produced by the board.

Process scheduling in traditional Unix operating systems cannot satisfy the real-time constraints of CM applications as described above. We illustrate by describing process scheduling in Solaris 2.3, where processes run in one of three *scheduling classes*: `RT` (real-time), `SYS`[1] (system) and `TS` (time-sharing). Priorities in a scheduling class are mapped to a set of *global* priorities. `RT` priorities are mapped to higher global priorities than `SYS` priorities, which are in turn mapped to higher global priorities than `TS` priorities. At any time, the system executes a runnable process with the highest global priority.

A user process in Solaris 2.3 runs in the `TS` class by default. There is a time quantum associated with every `TS` priority. Whenever a `TS` process uses up a time quantum, the system lowers the priority of the process. On the other hand, if a process has been blocked for a long time, the priority of the

[1] The `SYS` class is, however, not available to user processes and will not be considered further in this paper.

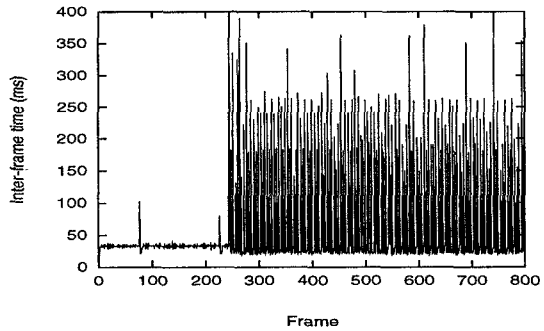

Figure 2: Times between pictures sent by a video application run as a Unix TS process.

process is raised. This approach provides fast response time to interactive applications without starving compute-bound applications. Moreover, a TS process is given a "kernel" priority whenever it blocks inside the kernel. The priority given depends on the condition on which the process is blocked. Hence, the priority of a TS process is dynamically changed by the system in an ad-hoc manner, and it cannot be used to specify an application's progress requirements. Fig. 2 illustrates how applications in TS class can fail to meet real-time constraints. In the figure, we show the times between pictures sent by a video application similar to the one described above. At first, pictures were mostly sent every 33.3 ms. After several compute-bound applications were started, however, interference from these other applications caused the video application to receive insufficient CPU time to keep up with the picture rate. Many pictures were skipped, and inter-frame times of 100 ms or more were common. (We will revisit this example in section 7 for the scheduler proposed in this paper.)

The RT scheduling class is intended to give users tighter control over how user processes are scheduled. RT priorities are never modified by the system, and an RT process always has priority over processes in the other scheduling classes. The RT class thus allows a user to run "performance critical" applications without interference by other system activities. However, like TS priority, RT priority lacks any QoS interpretation. A user must translate the progress requirements of applications to RT priorities in an ad-hoc manner. More importantly, the lack of QoS interpretation for RT priorities means that the system cannot do effective admission control. Without admission control, long term system overload cannot be prevented. Finally, since an RT process cannot be preempted by system processes, an RT process that does not voluntarily give up the CPU can block out all other system activities. When that happens, the only way for a system administrator to regain control of the workstation is to reboot the system.

1.1 Our contributions

For integrated CPU scheduling of CM and other applications, we propose the use of a family of *adaptive rate-controlled* (ARC) schedulers with the following properties: (1) reserved rates can be negotiated, (2) QoS guarantees are conditional upon process behavior, and (3) firewall protection between processes is provided. In this paper, we present a particular scheduler called RC together with two rate adaptation strategies. The CPU scheduling framework that uses RC allows applications to specify a reserved *rate* (between 0 and 1) and a time interval known as *period* (in μs). It provides the following progress guarantee: a "punctual" (a notion to be made precise in section 5.3) application with rate r and period p is guaranteed at least krp CPU time over time interval kp, for $k = 1, 2, \ldots$, where each interval is measured from when the application first becomes runnable. Although our framework is motivated by the requirements of CM applications, it is appropriate for scheduling other applications as well. This is desirable since, in a general purpose workstation environment, CM and other applications run together.

Our main contributions are 1) implementation of the scheduling framework and its integration into a workstation operating system, 2) an on-line scheduling algorithm that, in contrast to classical real-time scheduling algorithms, provides a progress guarantee to each process independent of the behavior of other processes, 3) empirical evaluation of our framework in scheduling CM and other applications, and 4) support for *rate adaptation* whereby the workstation kernel helps a user application adapt its current reserved rate by providing it with feedback information.

1.2 Related work

The case for an integrated scheduling policy for diverse applications has been advocated by other researchers, for example, [6]. However, not enough details of the algorithm are given in [6] for comparison with our approach. Rather than integrated scheduling, a three level hierarchical scheduler for a video-on-demand service has been proposed in [3].

The implementation of our scheduling framework is based on extending an existing operating system to support real-time scheduling. This is similar to the work of RT-Mach [8], which is an extension of the Mach operating system. Our requirement that a process's progress guarantee be protected from the execution behavior of other processes is similar in objective to the *processor capacity reserves* abstraction in [5]. There is, however, a key difference between processor capacity reserves and our solution, i.e., only scheduling algorithms with the firewall property are considered in our approach, thereby eliminating the need for an explicit monitoring mechanism to enforce firewall protection from interference. To illustrate, in RT Mach's implementation of processor capacity reserves, a reserve must be periodically replenished and an overrun timer must be set to expire at the time a process is supposed to voluntarily give up the CPU. Should the overrun timer expire, the reserved priority is *depressed* to an unreserved priority. In comparison, our system does not require such a mechanism for monitoring and policing, nor does it distinguish between reserved and unreserved priorities. ARC scheduling in our system is based upon a uniform class of dynamically computed priority values, one for each process.

Several rate-based algorithms with the firewall property have been proposed for scheduling packets in a network switch. Our algorithm is conceptually similar to the VC algorithm [9, 11] but with two differences needed for CPU scheduling: (1) a period parameter is introduced, and (2) in computing the priority value of a process, the expected finishing time of the previous work executed by the process is used instead of the expected finishing time of the work to be scheduled.

Several other packet scheduling algorithms have been designed to achieve various notions of fairness [1, 2]. These algorithms can also be used for ARC scheduling, with fairness achieved at the expense of more implementation overhead. However, our experimental results show that real-time video and audio applications are not greedy, and the notions of fairness as defined in [1, 2] is not an important concern for these applications.

1.3 Organization of this paper

In section 2, we discuss the classical rate-monotonic and earliest deadline first scheduling algorithms, and illustrate how a straightforward implementation of these algorithms in a general purpose workstation may lead to unsatisfactory results. In section 3, we relate the CPU scheduling work described in this paper to a new operating system architecture we proposed for supporting distributed multimedia applications. Section 4 introduces a rate-based reservation model for CPU time. The proposed scheduling framework, which consists of a priority based on-line scheduler, an admission control interface, a monitoring module and a rate adaptation interface, is described in section 5. Section 6 reports our experience in implementing the CPU scheduler in Solaris 2.3. Experimental results reported in section 7 show the effectiveness of our implementation for many test cases.

2 CLASSICAL REAL-TIME SCHEDULING

Many classical real-time scheduling techniques have been applied in multimedia operating systems [7]. Two algorithms that are generally believed to be suitable for scheduling CM applications are the rate-monotonic (RM) algorithm and the earliest deadline first (EDF) algorithm. We briefly review each of these algorithms [4].

Analysis of RM and EDF scheduling has made use of the following *periodic specification* for the execution of a process, say i. The specification has two parameters: a period, P_i (in seconds), and a computation time requirement per period, C_i (in seconds). An *event*, which requires C_i seconds of CPU time to process, is assumed to arrive at the beginning of each period. The *deadline* of an event, which is the time by which processing of the event must complete, is assumed to be the beginning of the next period. This model of execution is illustrated in Fig. 3.

The RM algorithm assigns the period P_i as a static priority value of process i. This priority value is interpreted such that the lower the value, the higher is the RM priority of the process. Liu and Layland [4] show that if $\Sigma_i C_i/P_i \leq n(2^{1/n} - 1)$, where the summation is over all processes in the system, then each process gets the following *progress guarantee*: For all i, the process will be scheduled to run for C_i within period P_i. This implies that the processing of each event will complete by its deadline. The condition $\Sigma_i C_i/P_i \leq n(2^{1/n} - 1)$ is the admission control criterion. When n is large, the right hand side is about 0.69. Hence, RM scheduling is in general not able to achieve 100% processor utilization. However, if the process periods are harmonic, then it can be shown that 100% processor utilization is indeed achievable.

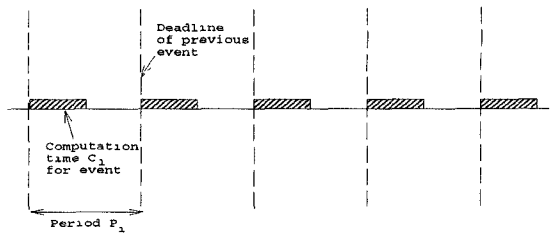

Figure 3: Periodic specification in classical real-time scheduling.

In contrast to RM, the EDF algorithm is a dynamic priority algorithm. At any time, the priority of a process is not fixed, but is determined by the deadline of its next event. A process with an earlier deadline value has a higher EDF priority. For EDF scheduling, it is proved that if $\Sigma_i C_i/P_i \leq 1$, then each process gets the same progress guarantee as RM scheduling [4]. Hence, unlike RM, full processor utilization is in general achievable with EDF.

Clearly, the progress guarantee by RM and EDF is useful in scheduling CM applications. For example, it can be used to schedule the video application described in section 1 such that each picture is processed before the next picture is produced by the video capture board. However, a straightforward implementation of either algorithm in a general purpose workstation environment may not yield satisfactory results. This is because the execution profile of a real application may not conform to the periodic specification. To illustrate, consider two processes, say Q and R, scheduled by the RM algorithm. Process Q has a period of 80 ms, and a per period computation requirement of 40 ms. Process R has a period of 40 ms, and a per period computation requirement of 20 ms. Since the periods are harmonic, the achievable processor utilization is 100% and is not exceeded in this example. Because R has a smaller period than Q, it has higher RM priority than Q. Now consider the execution profiles of the two processes shown in the top two rows in Fig. 4. Note that Q conforms to its periodic specification, whereas R does not. The row labeled "RM" shows how the processes are scheduled by RM. At the beginning of the first period (time 0), R is scheduled to run. RM does not require R to give up the CPU after running for 20 ms, and R goes on to run until 80 ms. The result is that Q is not scheduled at all during the first 80 ms. Hence, Q's progress guarantee is violated even though its execution conforms to its periodic specification.

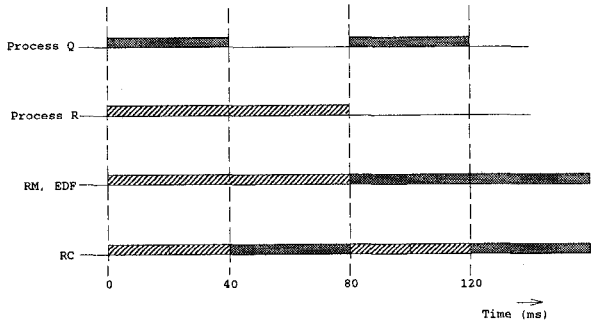

Figure 4: A "greedy" scheduling example.

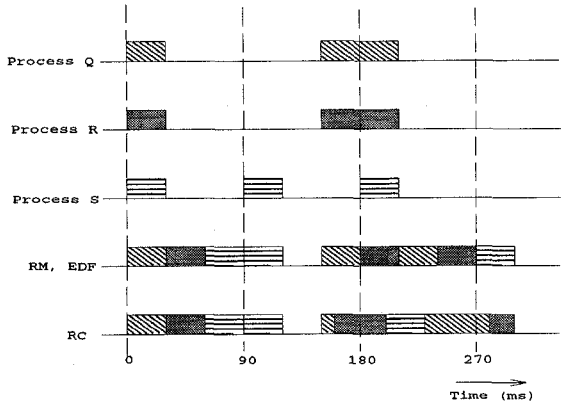

Figure 5: A "late" scheduling example.

EDF suffers from the same interference problem if the actual processing time of an event is longer than the processing time C_i assumed in admission control. For example, if the event that arrives for R at time 0 takes 80 ms to complete, EDF performs the same as RM in the example in Fig. 4.

The above example shows how a "greedy" process, such as R, that runs ahead of its periodic specification can affect the progress guarantees to other processes. However, scheduling requests that are late with respect to the periodic specification can also cause problem. Consider the scenario shown in Fig. 5. There are three processes, Q, R and S, each having period 90 ms and computation time requirement per period 30 ms. They are scheduled according to the RM alogrithm (see the row labeled "RM" in Fig. 5). Because the processes all have the same period and hence RM priority, the system has arbitrarily decided to schedule Q ahead of R, and R ahead of S. According to the periodic specification, both Q and R should request to run for 30 ms at the beginning of the second period (90 ms). However, Q and R are late and they do not become runnable until 150 ms. The result is that even though S becomes runnable at 180 ms, it cannot be scheduled until 270 ms. The progress guarantee of S is violated from 180 ms to 270 ms, despite the fact that the execution of S conforms to its periodic specification. EDF suffers from the same interference problem if event arrivals can be late. In our example, if the second events for Q and R arrive at 150 ms instead of 90 ms, then EDF performs the same as RM.

The scheduling framework proposed in this paper allows processes to reserve CPU time based on rates of progress. Moreover, we believe that it is desirable to provide *firewall protection* between processes, i.e., our system guarantees that each "punctual" (see section 5.3) process makes progress at its reserved rate *independent of the behavior of other processes.* Firewall protection is achieved by a form of *rate control* that will be made clear in section 5.

3 OS ARCHITECTURE OVERVIEW

We previously proposed an operating system architecture for supporting distributed multimedia [10]. The architecture makes use of *I/O efficient buffers* and a `fast_write()` system call to reduce the end-to-end latency of network data transfers. It also makes use of *kernel threads* for reduced system calls and rate-based flow control. The system was prototyped as an extension to Solaris 2.3, and the CPU scheduling framework reported in this paper has since been integrated into the prototype system. In this section, we describe features of the prototype system that are relevant to CPU scheduling.

First, in contrast to a traditional Unix kernel, the Solaris 2.3 kernel is fully preemptible except for a few short protected intervals. This is critical for us to obtain good real-time application performance, since it allows a high priority process to preempt a lower priority process even though the latter may happen to be in the middle of a long duration system call. Second, the Solaris kernel implements priority inheritance for most synchronization primitives such as semaphores and mutex locks. This prevents a high priority process from being blocked indefinitely by lower priority processes because of lock contention.

Third, in our prototype system, a lightweight kernel thread can be used to multiplex a shared network connection among multiple user processes [10]. Specifically, a user process with packets to send enqueues the packets to a send control queue. A kernel thread is then responsible for moving packets from the send control queue to a network interface queue at a reserved bit rate (see Figure 6). The kernel thread also performs rate-based flow control of shared access to a network connection.

There are several timing constraints in process scheduling: a user process must be scheduled such that it can enqueue packets to the send control queue "in time", and the kernel thread must be scheduled such that it can move the packets to the network interface queue "in time". As described in [10], the timeliness condition for a kernel thread means that the kernel thread will be periodically scheduled with a maximum CPU time per period. Fig. 6 shows the relationship between CPU scheduling and send side packet scheduling by a lightweight kernel thread.

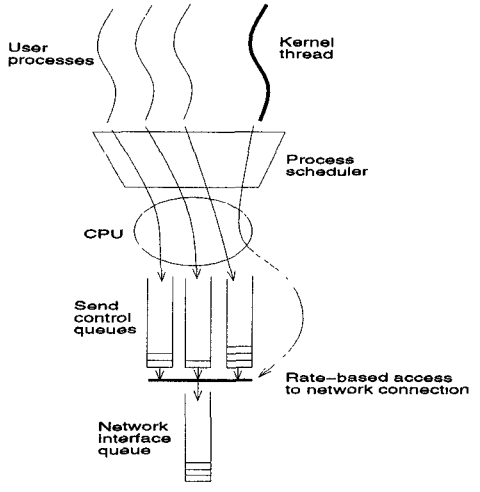

Figure 6: Relation of CPU scheduling to our OS architecture.

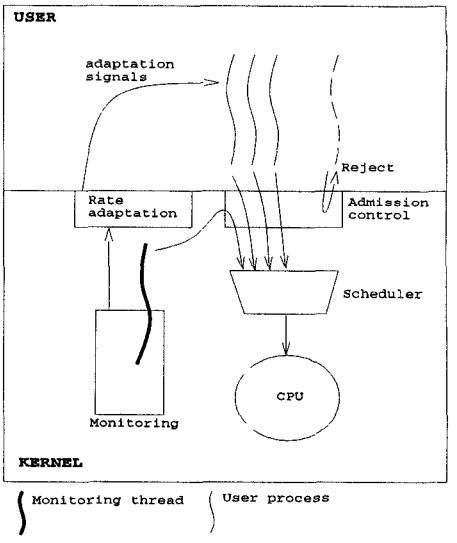

Figure 7: Scheduling framework.

4 RATE-BASED RESERVATION

Our system allows processes to reserve CPU time based on a *rate*[2] of progress, r $(0 < r \leq 1)$, and a time interval p in μs known as *period*. The rate can then be viewed as a guaranteed fraction of CPU time that a "punctual" (this notion will be made precise in section 5.3) process will be allocated over time. Specifically, the process will be allowed to run for at least krp time over time interval kp for $k = 1, 2, \ldots$, measured from when the process first becomes runnable. For example, if the rate is 0.5 and the period is 100 ms, then the process will be allowed to run for at least 50 ms over the first 100 ms since the process first becomes runnable, for at least 100 ms over the first 200 ms, etc.

The rate-based reservation model is similar to the periodic specification in section 2, by considering C_i/P_i to be the rate of process i. If the execution of a process does conform to the periodic specification, then the rate-based model ensures the same progress guarantee to the process as RM and EDF scheduling. However, there are two important differences. First, our system provides firewall protection between processes such that the progress guarantee to a process is independent of the behavior of other processes. Second, the rate-based model makes explicit the notion of a guaranteed rate of progress, which we believe is natural even for applications that are not "real-time" and not inherently periodic. For example, consider a numerical analysis application that solves a system of linear equations. For a particular problem, the application is continuously enabled (meaning that it is always ready to run) and takes 5 seconds of CPU time to complete. Suppose the user runs the application with a rate of 0.01 and a period of 0.5 seconds. In the absence of competing processes, the system will allow the application to run continuously for 5 seconds and terminate. On the other hand, if the system is highly loaded, the system will still ensure that the application will run at least $5k$ ms for every $0.5k$ seconds after the application first becomes runnable, and the application will terminate in at most 500 seconds.

[2] Unless otherwise specified, we shall use the term *rate* to mean *reserved* rate.

5 SCHEDULING FRAMEWORK

An overview of our scheduling framework is shown in Fig. 7. The framework consists of the following components. First, there is an on-line scheduler that schedules processes according to dynamic rate-controlled priority values (hereafter called RC values) to be defined in section 5.1. Second, there is an admission control interface that admits or rejects new processes based on the rate-based reservation model in section 4. Admission control limits system overload so that rate guarantees to processes can be met. Third, a monitoring module and a rate adaptation interface allow processes to adjust their reserved rates based on feedback information from the kernel.

5.1 On-line scheduler

Having characterized CPU reservation with a rate r $(0 < r \leq 1)$ and a period p (in μs) in section 4, we next present a rate-controlled (RC) on-line process scheduler. RC schedules processes according to a per-process RC value computed by algorithm RC specified in Fig. 8. In the specification, Q is the process for which the algorithm is executed, $curtime$ (in μs) is the time at which the algorithm begins execution,[3] $p(Q)$ and $r(Q)$ denote, respectively, the period and rate of Q's CPU reservation, and $val(Q)$ denotes the RC value of Q. In addition, two per-process state variables are maintained: $start(Q)$ (in μs and initialized to the time at which Q first becomes runnable) and $finish(Q)$ (in μs and initialized to 0). $start(Q)$ is immutable and so always gives the time at which Q first becomes runnable. $finish(Q)$ keeps track of the *expected finishing time* of the previous computation performed by Q.

Intuitively, the expected finishing time is the time when the

[3] More precisely, $curtime$ is the time of the rescheduling event (explained below) that causes the algorithm to be executed.

Algorithm RC(Q)

1. **if** (Q changes from blocked to runnable)
2. $finish(Q) := \max(finish(Q), curtime)$;
   else
3. $runtime$:= total time process Q has run since RC was last executed for Q;
4. $finish(Q) := finish(Q) + runtime/r(Q)$;
   fi;
5. **if** (Q is not blocked)
6. Find k s.t. $start(Q) + (k-1) \times p(Q) \leq finish(Q) < start(Q) + k \times p(Q)$;
7. $val(Q) := start(Q) + k \times p(Q)$;
   fi;

Figure 8: Specification of Algorithm RC.

previous computation would complete had the computation proceeded at rate $r(Q)$. For example, if some computation took 1 second of CPU time and the rate is 0.1, then the expected finishing time is $1/0.1 = 10$ seconds from the start of the computation. Notice, however, that when Q becomes runnable, if the current real time ($curtime$) is later than $finish(Q)$, $finish(Q)$ will be updated to $curtime$. Because of this, a process that has not run for a long time and has not been using its reserved rate, would not get a very low RC value when it becomes runnable.

To understand how $val(Q)$ is computed, think of the lifetime of Q as starting at $start(Q)$ and subsequently divided into periods of $p(Q)$ each. Then, if the expected finishing time of the previous computation performed by Q falls within the kth period, $val(Q)$ is set to the end of the kth period (hence an RC value is an expected time value).

To describe when algorithm RC executes, define a *rescheduling point* to be the time when one of the following events occurs: 1) the currently running process becomes blocked, 2) a system event occurs that causes one or more processes to become runnable, or 3) a periodic clock tick[4] occurs. At a rescheduling point, the RC value of *some* of the processes may change, and algorithm RC needs to be executed for only these processes. Specifically,

- When the currently running process becomes blocked, RC is executed for it.
- When a system event occurs that causes one or more processes to become runnable, RC is executed for each process that becomes runnable.
- When a periodic clock tick occurs in the system, RC is executed for the currently running process if one exists.

After RC values have been recomputed for these affected processes at a rescheduling point, a runnable process with the smallest RC value is chosen for execution; ties are broken in favor of the currently running process, and otherwise arbitrarily.

[4]The period of this clock tick is a system wide parameter (1 ms in our prototype) and not to be confused with the period of a process.

Time	Process Q		Process R		
(ms)	$finish$	val	$finish$	val	Scheduled
0	0	80	0	40	R
20	0	80	40	80	R
40	0	80	80	120	Q
80	80	160	80	120	R
100	80	160	120	160	R
120	80	160	160	200	Q

Table 1: Illustration of algorithm RC for "greedy" scheduling example in Fig. 4.

Note that RC is a dynamic priority scheduling algorithm. We view RC value recomputation as a form of *rate control*. First, the priority of a process that tries to run ahead of its reserved rate will be lowered at a clock tick and the process may be forced to yield the CPU. Second, as we mentioned, a process that has not been using its reserved rate will not get a very low RC value (hence a very high priority) when it becomes runnable. In other words, unused "credit" cannot be saved by a process.

5.2 Examples revisited

The effects of rate control can be illustrated by revisiting the scheduling examples in Figures 4 and 5. For these examples, we assume a clock period of 10 ms, and a clock tick occurs at $0, 10, 20, \ldots$ ms. In Fig. 4, process Q has rate 0.5 and period 80 ms while process R has rate 0.5 and period 40 ms. The row labeled "RC" shows how Q and R are scheduled by RC. At time 0, both processes first become runnable. Hence $start(Q) = start(R) = 0$. Table 1 shows the values (in ms) of the scheduling variables at various times when the RC value of either process changes. From Fig. 4, it can be seen that both Q and R get their progress guarantees.

Now consider the scheduling example in Fig. 5. All of the processes, Q, R and S, have rate 0.33 and period 90 ms. $start(Q) = start(R) = start(S) = 0$. Table 2 shows the values (in ms) of the scheduling variables at various times when the RC value of any process changes. In the table, "-" means that the value of the variable does not matter since the corresponding process is blocked. The tie-breaking rule of arbitrarily selecting a runnable process with the lowest RC value for execution is invoked at times 0 ms, 30 ms, 150 ms and 200 ms. Notice from Fig. 5 that S gets its progress guarantee with RC scheduling. However, Q and R do not get their progress guarantees because they are late.

Note that there is an inherent tradeoff between a more predictable performance and a smaller overhead for rate control. In our system, rate control occurs at each clock tick. For firewall protection and predictable performance, a small clock tick is desired, but the higher the clock frequency, the higher the rate of RC value recomputation. However, notice that 1) RC value recomputation is very simple and only needs to be done for the currently executing process at a clock tick, and 2) most of the time, the RC value of the currently executing

Time	Process Q		Process R		Process S		
(ms)	$finish$	val	$finish$	val	$finish$	val	Sched
0	0	90	0	90	0	90	Q
30	90	180	0	90	0	90	R
60	90	180	90	180	0	90	S
90	90	180	90	180	90	180	S
120	-	-	-	-	-	-	none
150	150	180	150	180	-	-	Q
160	180	270	150	180	-	-	R
170	180	270	180	270	-	-	R
180	180	270	210	270	180	270	R
200	180	270	270	360	180	270	S
230	180	270	270	360	-	-	Q
260	270	360	270	360	-	-	Q
280	-	-	270	360	-	-	R

Table 2: Illustration of algorithm RC for "late" scheduling example in Fig. 5.

process will not change, and hence processes do not need to be rescheduled. For example, both RM and RC require 8 context switches to schedule the processes in Fig. 5.

5.3 Admission control

To satisfy the real-time requirements of user processes, CPU time cannot be oversubscribed. Hence, admission control is an essential component of our scheduling framework. When a new process is created, the system checks whether enough CPU capacity exists to satisfy the rate request of the new process, without violating the guarantees to processes that are already admitted. The admission control criterion we use is $\Sigma_i r_i \leq 1$, where r_i is the rate of process i. We motivate this admission control criterion by considering an *idealized execution environment* in which the period of clock tick is infinitesimally small, and the overhead of rate control is zero. There are n processes, $Q_1, \ldots, Q_n$, in the system. For $i = 1, \ldots, n$, Q_i runs with rate r_i and period p_i. Consider some process Q_j. For simplicity of exposition, the time at which Q_j first becomes runnable is time 0 in the statements of Definition 1 and Theorem 1.

Definition 1 *Q_j is* punctual *if it generates at least $(k+1)r_j p_j$ seconds of work over time interval $[0, kp_j]$, for $k = 0, 1, \ldots$.*

Theorem 1 *If Q_j is punctual and $\Sigma_i r_i \leq 1$, then Q_j is scheduled by RC to run for at least $(k+1)r_j p_j$ time over time interval $[0, (k+1)p_j]$, for $k = 0, 1, \ldots$.*

A proof of Theorem 1 is given in the Appendix. In the proof, "Ln" refers to the line of code labeled n in Figure 8. Clearly, the idealized execution environment is not realizable in practice. It can only be approximated. However, the experimental results in section 7 show that the RC scheduler performs as intended in a real workstation environment.

5.4 Rate adaptation

The reservation model introduced so far assumes a rate that is fixed for the lifetime of a process. This assumption may be overly restrictive for a dynamic execution environment. Indeed, when an application is started, a user may not know the appropriate rate to use. First, the user may have insufficient knowledge of the application. Second, the application may not have a constant rate of execution due to, for example, the application's inherent characteristics (scene changes may cause a video playback application to run with different rates at different times) or the application's need to adapt to changes in the environment (e.g. to cooperate with network flow control). When a process runs far behind its reserved rate, any unused CPU time will not be available for reservation and CPU utilization decreases. On the other hand, if a process runs far ahead of its rate, its priority will be lowered by RC rate control, which may later adversely affect its real-time performance.

In view of the above, our system provides a *rate adaptation* mechanism, whereby the kernel helps a user process determine its rate by providing feedback information on the process's execution. Rate adaptation enables a process to react to medium to long term changes in the process's execution rate (such as on the order of tens of seconds or longer). It consists of a monitoring module that monitors process execution, and a rate-adaptation interface between the kernel and user processes.

To enable rate adaptation, a process has to register with the system. A monitoring thread running with a period of m seconds ($m = 2$ in our current system) monitors the execution of registered processes. We are interested in two quantities. The first one, called the *lag* (in μs), measures how far ahead a process is running of its reserved rate at time t. Note that if a process, say Q, is running ahead of its rate, $finish(Q)$ will get farther and farther ahead of real time. Hence the lag of a process at time t is defined to be $\max(finish(Q) - t - p(Q), 0)$. The second quantity, called the *lax* (in %), measures the percentage of reserved CPU time unused by the process during the last monitoring interval T (i.e. the time interval between the current and the last monitoring). It is defined as $\max(100(1 - runtime/(rT)), 0)$, where $runtime$ (in μs) is the total time the process has run during the time interval. We expect the rate adaptation mechanism to be used only by CM applications that have a fairly constant rate of progress over a monitoring interval.

The system informs a process of "significant" mismatches between the reserved rate and the current execution rate. For this purpose, a process specifies two parameters to the system when registering for rate adaptation: a *lag tolerance* (in μs) and a *lax tolerance* (in %). When monitored, if the process has a maximum lag over the last monitoring interval that is greater than the lag tolerance, a signal to increase rate is sent to the process. Also, if the lax of the process over the last monitoring interval is x and higher than the lax tolerance, a signal to slow down by x% is sent to the process. The signals to speed up and slow down are know as *rate adaptation signals*. The application installs a signal handler to react to rate adaptation signals in an application specific manner (section 7 describes two strategies an application might use).

6 IMPLEMENTATION

Our scheduling framework has been implemented in Solaris 2.3. For the on-line scheduler described in section 5.1, we added a new scheduling class RC. Most of the RC class specific code is implemented as a loadable module that can be dynamically linked with the rest of the kernel. The class independent scheduling code in Solaris already has hooks that call the RC code at certain strategic points. However, we have found it necessary to modifiy the original kernel in three respects. First, we added a hook, which we name CL_RESUME, for class specific code to run when a process is "resumed" (i.e. CL_RESUME is inserted before each call to `resume()`, the kernel call to switch the CPU to a new process). CL_RESUME allows the system to know when a process is allocated the CPU, and hence to monitor how long the process has run. Second, process priority in Solaris has type `pri_t`, which is simply defined as `short`. `pri_t` is, however, not consistently used throughout the kernel. Certain code uses variables of type `long` interchangeably with variables of type `pri_t`. We have found it necessary to define a new `pri_t` type with type specific methods for, say, initializations and comparisons. We also removed the intermixing of `pri_t` variables with variables of other types. Third, processes that share the same dispatch queue[5] in Solaris have the same dispatch priority and are mostly served in a round robin manner. In the modified kernel, all processes in the RC class share a global dispatch queue, and processes have to be queued in RC value order.

In addition, we made two significant changes to our system configuration. First, we shortened the clock interrupt interval from 10 to 1 ms. This gives a finer granularity of control with a small performance penalty. Second, we run all system threads (except threads for interrupt processing) in the RC class. System threads in Solaris 2.3 are used for a variety of purposes such as starting asynchronous read-aheads in file systems, processing callouts, reaping freed system resources, and background processing of stream service routines. To allow all system activities to continue to make non-zero progress despite the demand of user applications, we have assigned each system thread a rate of 0.002 and a period of 200 ms. Such an assignment is admittedly ad-hoc and user applications cannot rely on it for performance guarantees. Of particular concern are system threads used in the stream subsystem, since networking access is an integral part of any distributed CM application. In the system architecture proposed in [10], however, we assume that network protocols are implemented in user space, rather than as stream modules, and the kernel thread used for flow control has well-defined scheduling parameters (i.e. period of execution and computation requirement per period).

7 EXPERIMENTAL RESULTS

We have performed a large number of experiments to evaluate the effectiveness of our scheduling framework. Before we discuss individual experiments, we make the overall, qualitative observation that user applications running in RC never caused control over the system to be lost. In particular, shell commands could still be started and processes could be killed (we used a `tcsh` shell with a rate of 0.002). This is in contrast to the RT class in Unix SVR4, where a "greedy" RT process that never gives up the CPU can effectively "take over" the entire workstation and force a system reboot.

[5] A dispatch queue is a queue of runnable processes, or processes eligible for *dispatch*.

Prog	Param	Description
`greedy`	`n`	Repeats rounds of following computation: $y = \sin(x)$. After the nth round, the time taken for the first `n` rounds is printed. This represents a compute-bound application.
`video`	`[-d]`	A video server that repeatedly reads a Cell-B compressed picture from the SunVideo `rtvc` device and sends each picture (encapsulated by an application level protocol) to a UDP connection. If `-d` is specified, each picture is additionally Cell-B decompressed in software and displayed in an X window. A frame rate of 30 fps is achievable on our workstation.
`audio`	-	An audio server that does radio broadcast in a local area network. It captures PCM encoded audio at 64 kbps from a local audio device, encapsulates each audio sample by an application level protocol, and sends the sample to a UDP connection. We have configured the audio device to return samples every 20 ms.
`X`	-	An X window system server that handles display in an X window.
`sema`	-	Repeats rounds of following execution: enters a semaphore protected critical section, does some computation, exits the critical section, and does some more computation.
`shell`	-	A `tcsh` shell command interpreter.

Table 3: Test suite of applications.

7.1 Test suite

In our experiments, we used the test suite of applications shown in Table 3. We chose the applications to have characteristics representative of common applications for a general purpose workstation. For example, `video` and `audio` are CM applications, `shell` is a traditional interactive application, and `greedy` is a batch like, compute-bound application. Table 4 summarizes the experiments that were performed.

7.2 Test cases

simple This simple experiment shows that our scheduling algorithm in fact allows applications to make progress at their

Case	Program	Rate	Period (ms)
simple	greedy 3000000	0.27	50
	greedy 3000000	0.63	50
lock-a	sema	0.09	80
	sema	0.18	80
	sema	0.63	80
lock-b	sema	0.18	80
	sema	0.27	80
	sema	0.45	80
aud-g3	audio	0.15	20
	3xgreedy 3000000	0.09	10
vid-g3	video -d	0.65	34
	X	0.05	34
	3xgreedy 3000000	0.1	10
vid-gx	video -d	varied	34
	X	0.05	34
	greedy 1000000	varied	30
av-g3	audio	0.15	20
	video -d	0.6	34
	X	0.05	34
	3xgreedy 3000000	0.04	10
ra-vg	video -d	adaptive	34
	X	0.05	34
	greedy 1000000	?	30

Table 4: Cases of experimental runs.

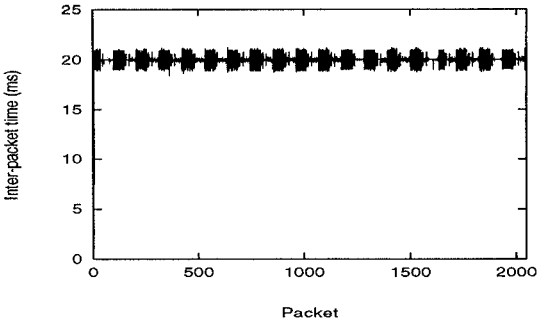

Figure 9: Profile of inter-packet times when audio started while three processes executing greedy 3000000 were running.

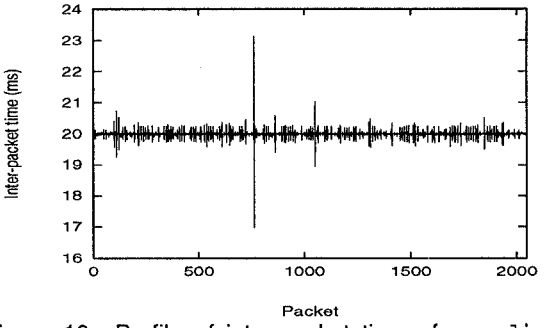

Figure 10: Profile of inter-packet times for audio when greedy 3000000 started about 1 minute after audio (trace started several seconds before greedy started).

reserved rates of execution. When run by itself (i.e. with minimal competition from other processes), greedy took 19.99 seconds to complete 3000000 rounds of computation. In simple, the two processes with relative rates 0.7 : 0.3 took 28.64 and 67.20 seconds, respectively. It is straightforward to show that the higher rate process got roughly 69.76% (19.99 / 28.64) of CPU, whereas the lower rate process got 29.73%.

lock-[ab] The experiments lock-a and lock-b tested the effects of lock contention, as each process has a critical section guarded by the same semaphore. We measured the time taken for each process running sema in Table 3 to complete 19 rounds of execution. In lock-a, the processes with relative rates 0.7 : 0.2 : 0.1 took, respectively, 17.48, 60.62 and 121.06 seconds. The measured ratios of execution times are thus 1 : 3.47 : 6.93 and are close to the expected ratios of 1 : 3.39 : 7.00. In lock-b, the processes with relative rates 0.5 : 0.3 : 0.2 took 24.03, 39.99 and 60.35 seconds respectively. The measured ratios of execution times are thus 1 : 1.66 : 2.51 and comparable to the expected ratios of 1 : 1.67 : 2.50.

aud-g3 Set up to send an audio packet every 20 ms, audio has arguably the most stringent timeliness requirement among applications in our test suite. We are therefore interested in knowing how well we can schedule audio to meet its timing constraints when we have concurrently running CPU intensive applications. In particular, we would like audio to be able to send each 20 ms sample of audio data before the next sample has been produced by the audio device. In our experiment, we first started 3 RC processes, each running greedy 3000000 with a rate of 0.1. Then we started audio with rate 0.15. To quantify the "timeliness" of audio, we recorded a 41 second trace of the *inter-packet times* (i.e. the times between sends of consecutive audio packets). The trace is shown in Fig. 9. The maximum inter-packet gap is 21.59 ms, remarkably close to the ideal value of 20 ms.

We also performed a variant experiment of aud-g3, in which we examined whether the timeliness of audio will be adversely affected if we start greedy *after* audio has been running steadily. In our experiment, greedy 3000000 was started about 1 minute after audio. The 1 minute lead time gives the actual execution rate of audio to stabilize after a significantly more CPU intensive phase of program startup. The trace of inter-packet times is shown in Fig. 10 (we started the trace several seconds before greedy started). The maximum inter-packet time is 23.16 ms.

vid-g3 For a video frame rate of 30 fps, video is expected to run and send the packets of each picture every 33.33 ms. Although this delay requirement is somewhat less stringent than audio, video is significantly more CPU intensive. In this experiment, we examined whether video is able to meet its timing constraints when run concurrently with other CPU intensive RC processes. We first started 3 processes each running greedy 3000000 with a rate of 0.09. Then we started video -d with rate 0.65. video communicates with the local X window system server through a Unix domain socket. X was run with rate 0.05. We traced the *inter-*

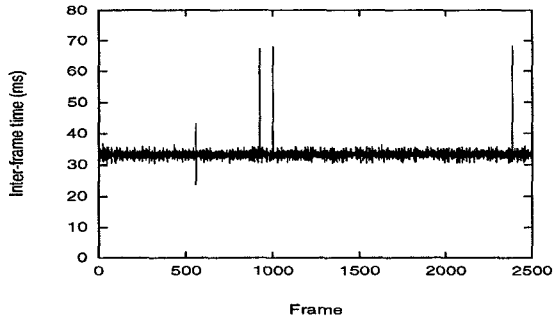

Figure 11: Profile of inter-frame times for `video`, when `video` was started while three processes executing `greedy 3000000` were running.

video rate	greedy rate	greedy time(ms)	greedy actual rate
0.4	0.5	11857	0.57
0.5	0.4	15416	0.44
0.6	0.3	21607	0.31
0.7	0.2	22471	0.30

Table 5: Execution time printed by `greedy 1000000` and actual execution rate of `greedy` with a competing `video -d` at various reserved rates (experiment `vid-gx`).

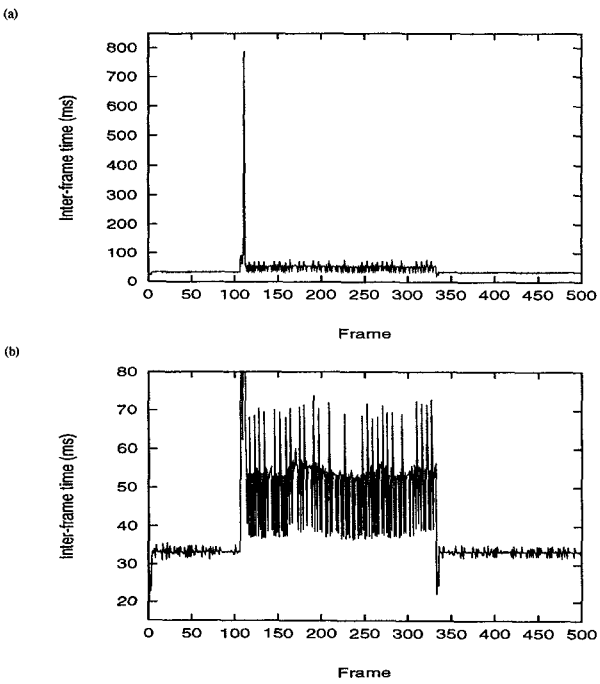

Figure 12: (a) Profile of inter-frame times for `video`, when `video` ran with a low rate of 0.4. (b) A magnified view showing the reduced frame rate.

frame times (i.e. the times between sends of *first* packets of consecutive video frames) for 2499 frames in Fig. 11. There were 3 deadline misses (a deadline miss occurs when a frame is dropped because `video` fails to process it in time). The misses occurred after frames 922, 999 and 2384, respectively, in the trace. However, these few misses do not suggest the existence of any weakness in our scheduling algorithm. We report that in another experiment in which we ran `video -d` just by itself, we still observed 4 deadline misses.

vid-gx This set of experiments investigates the progress rate of `greedy` as it runs against `video -d` at various reserved rates. In each experiment, `video` was started followed by `greedy 1000000` after a few seconds. The reserved rates of `video` and `greedy` were varied as in Table 5. In each case, we noted the actual execution time `greedy` printed after 1000000 rounds of execution. Dividing this actual execution time into 6759 ms (execution time `greedy 1000000` prints out when run by itself) yields the actual execution rate. The actual execution times and rates are reported in Table 5. Notice that the actual execution rate of `greedy` is consistently higher than the reserved rate. This is because the other pocesses in the system (e.g. X) did not make full use of their reserved rates. When `greedy` had a reserved rate of 0.3, 0.4 or 0.5, it had to compete with `video` for the "slack" CPU capacity left by the other processes. When this happens, the higher the reserved rate of `greedy`, the larger the fraction `greedy` took up of the slack capacity. When `greedy` had reserved rate 0.2, it nevertheless got an execution rate of 0.3. This is because `video` with rate 0.7 did not require much of the slack bandwidth.

As for `video`, it suffered minimal loss in performance when its reserved rate was 0.6 or 0.7. However, when its reserved rate was too low, such as 0.4, `video` clearly had to skip more pictures while `greedy` was simultaneously running. Fig. 12a profiles the inter-frame times for `video`, when `video` ran concurrently with `greedy` at a rate of 0.4. A large gap (about 0.8 second) is observed when `greedy` started. This is because `video` had been running significantly ahead of its reserved rate and was forced to slow down by the competing `greedy` process (in experiment `ra-vg`, we discuss how a user application can make use of rate adaptation to avoid this "punishment phenomenon"). After the initial gap, `video` continued to run with a lower frame rate (see Fig. 12b, a magnified view of Fig. 12a).

av-g3 We ran all of `audio`, `video` and `greedy` together in this experiment. First, 3 RC processes running `greedy 3000000` were started with a rate of 0.004, then `video -d` was started with rate 0.6 and finally, `audio` was started with rate 0.15. Fig. 13a shows a 50 second profile of the inter-packet times for `audio`. The jitters in scheduling were such that processing of alternate audio samples could be delayed until close to the time at which the next sample was produced. However, none of the packets missed its deadline. The maximum inter-packet gap was 37.37 ms. For `video`, the profile of inter-frames times is shown in Fig. 13b. There were 5 deadline misses during the 2485 frame trace. The maximum inter-frame time was 81.35 ms.

ra-vg We study whether applications can benefit from rate adaptation in this set of experiments. We experimented with two strategies that applications might use.

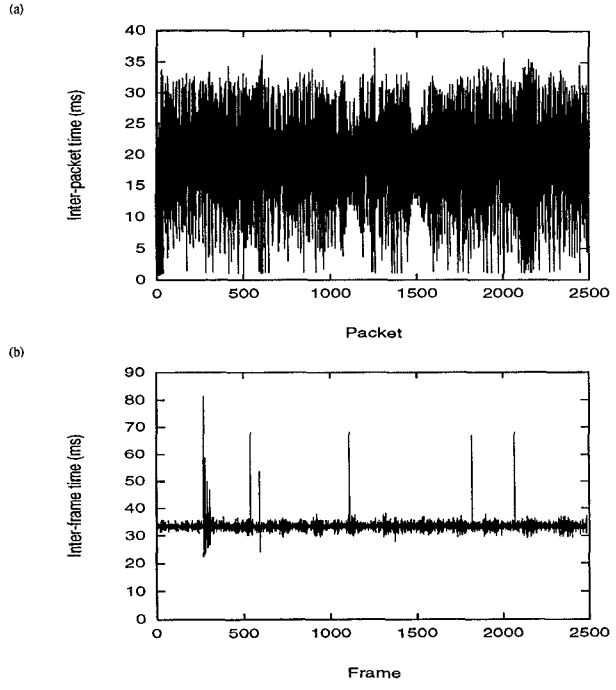

Figure 13: Profile of (a) inter-packet times for `audio` and (b) inter-frame times for `video` in experiment `av-g3`.

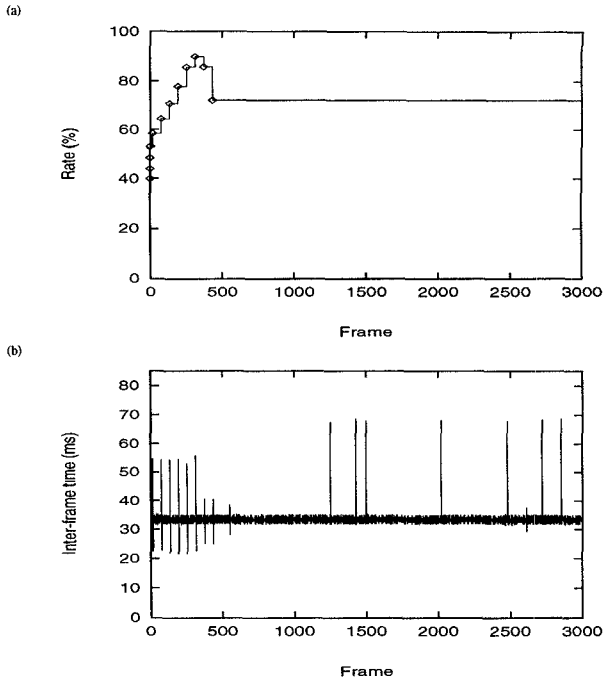

Figure 14: Profile of (a) rates and (b) inter-frame times for `video` with rate adaptation from an initial rate of 0.4.

In the first strategy, an application initially guesses a rate at which it should run, and then relies on rate adaptation to adjust its current rate upward or downward. In our experiment, `video` used an initial rate of 0.4, a lag tolerance of 34 ms and a lax tolerance of 10%. It adjusted its rate as follows: Upon receiving a signal to speed up, `video` increased its current rate by 0.1; upon receiving a signal to slow down by $x\%$, `video` decreased its rate by $(x - 5)\%$. The profile of rates at which `video` ran is shown in Fig. 14a. Note that after an initial *adaptation phase* in which `video` "hunted" for a stable rate to use, the rate stabilized at 0.721 at frame 435. The effects of rate adaptation on the inter-frame times are shown in Fig. 14b. During the adaptation phase, a frame was delayed by close to one frame time about every 2 seconds. This is because `video` needed to handle the rate adaptation signal about every 2 seconds. `video` achieved full performance after its rate had stabilized. In particular, even though we started a process running `greedy 1000000` shortly after frame 435, `video` managed to send a frame about every 33.33 ms. This is in contrast to the situation shown in Fig. 12, in which we observe a 0.8 second inter-frame time because `video` was started with a low rate of 0.4. There are totally 7 deadline misses in the 3000 frame trace.

We also examined a second strategy for rate adaptation in which an application starts with a very high rate and then relies on rate adaptation to adjust its current rate downward. In our experiment, `video` was started with an initial rate of 0.9, a lag tolerance of 34 ms and a lax tolerance of 10%. Upon receiving a signal to slow down by $x\%$, it decreased its rate by $(x - 5)\%$. Using this strategy, `video` had a single adjustment of its rate to 0.732 at frame 137 (Fig. 15a). The profile of inter-frame times in Fig. 15b shows that full performance was achieved throughout. In particular, starting `greedy 1000000` shortly after frame 137 and seconds before frame 3000 had no observable effects on the inter-frame times. There were totally 6 deadline misses in the 3000 frame trace.

8 CONCLUSION

We have presented a framework for integrated scheduling of CM and other applications in a general purpose workstation environment.

REFERENCES

1. J. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *Proc. INFOCOM '96*, pages 120–128, San Francisco, CA, March 1996.
2. Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queuing algorithm. In *Proceedings of ACM SIGCOMM '89*, pages 3–12, August 1989.
3. K.K. Ramakrishnan et al. Operating system support for a video-on-demand service. *Multimedia Systems*, 1995(3):53–65, 1995.
4. C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *JACM*, 20(1):46–61, 1973.
5. C.W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. IEEE Int'l Conf on Multimedia Computing and Systems*, Boston, MA, May 1994.

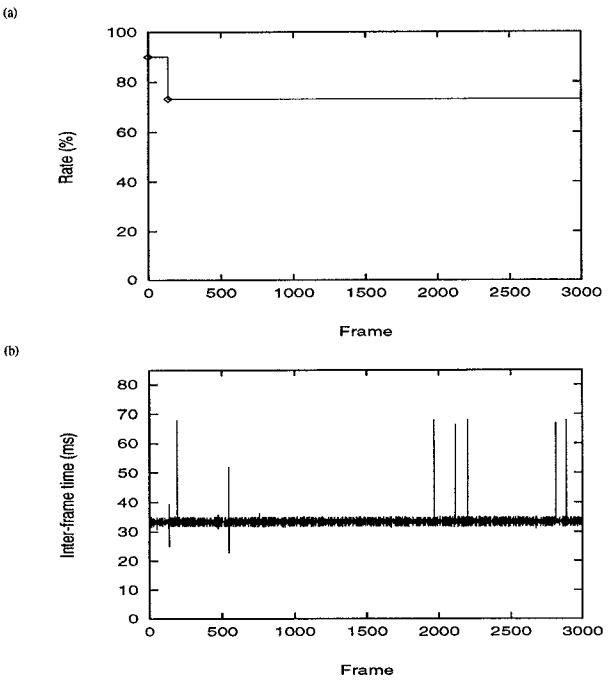

Figure 15: Profile of (a) rates and (b) inter-frame times for `video` with rate adaptation from an initial rate of 0.9.

6. Jason Nieh and Monica S. Lam. Integrated processor scheduling for multimedia. In *Proc. 5th International Worshop on Network and Operating System Support for Digital Audio and Video*, pages 215–218, Durham, NH, April 1995.
7. Ralf Steinmetz. Multimedia operating systems. *IEEE Multimedia Magazine*, 1995.
8. H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Toward a predictable real-time system. In *Proc. USENIX Mach Workshop*, October 1990.
9. Geoffrey G. Xie and Simon S. Lam. Delay guarantee of Virtual Clock server. *IEEE/ACM Trans. on Networking*, 3(6):683–689, December 1995.
10. David K.Y. Yau and Simon S. Lam. An architecture towards efficient OS support for distributed multimedia. In *Proc. IS&T/SPIE Multimedia Computing and Networking Conference*, January 1996.
11. Lixia Zhang. VirtualClock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.

Appendix: Proof of Theorem 1

We prove Theorem 1 by induction on k. **Base step.** For $k = 0$, since Q_j is punctual, it generates at least $r_j p_j$ seconds of work at time 0. To prove by contradiction, suppose this amount of work did not finish by time p_j. For this to happen, the CPU must have been occupied with work throughout the time interval $[0, p_j]$. Moreover, by the assumption that the period of clock tick is infinitesimally small, this work must have been scheduled with RC value not greater than p_j. There are two possible cases.

Case 1. In the busy period containing p_j, only work with RC value not greater than p_j was executed by time p_j. In this case, let $t < 0$ be the start of the busy period (i.e. the CPU was idle at time t^- but was doing work with RC value not greater than p_j throughout $[t, p_j]$). Because the CPU was idle at t^-, if any process, say Q_i, became runnable in $[t, p_j]$, the conditional test in L1 of Figure 8 would be true. L2 then ensures that Q_i's initial work in $[t, p_j]$ would not have received an RC value less than t. Because the RC value of any process is nondecreasing, we conclude that any work scheduled in $[t, p_j]$ had RC value at least t. By L4, L7 and the assumption that the period of clock tick is infinitesimally small, the maximum amount of work that can be scheduled for Q_i in $[t, p_j]$ is $\lfloor (p_j - t)/p_i \rfloor r_i p_i$.

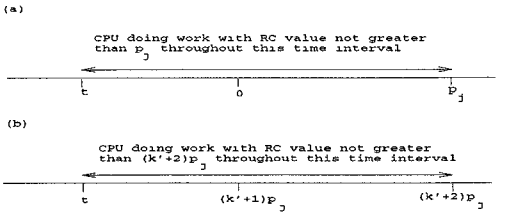

Figure 16: In (a), t is the time at which the CPU was last idle or a piece of computation with RC value greater than p_j last finished execution before time p_j. In (b), t is the time at which the CPU was last idle or a piece of computation with RC value greater than $(k'+2)p_j$ last finished execution before time $(k'+2)p_j$.

Case 2. In the busy period containing p_j, some work with RC value greater than p_j was executed before time p_j. In this case, let $t < 0$ be the time at which the last piece of work with RC value greater than p_j finished execution in the busy period. Consider any process Q_i. If Q_i was runnable at t^-, its RC value at t must be greater than p_j, since a piece of work with RC value greater than p_j finished execution at t. Hence no work was executed for Q_i in $[t, p_j]$. If Q_i was blocked at t^-, then, by L1 and L2, any work that might have been scheduled for Q_i in $[t, p_j]$ must have RC value at least t. By L4, L7 and the assumption that the period of clock tick is infinitesimally small, the maximum amount of work that can be scheduled for Q_i in $[t, p_j]$ is $\lfloor (p_j - t)/p_i \rfloor r_i p_i$.

The two cases are summarized in Fig. 16a. In either case, because the work of Q_j did not finish by p_j, we have

$$\begin{aligned} & \Sigma \lfloor (p_j - t)/p_i \rfloor r_i p_i > p_j - t \\ \Longrightarrow \quad & \Sigma (p_j - t) r_i > p_j - t \\ \Longrightarrow \quad & \Sigma r_i > 1, \text{since } p_j > t \\ \Longrightarrow \quad & \text{contradiction} \end{aligned}$$

Inductive step. Assume that Theorem 1 is true for $k = k' \geq 0$, i.e., the first $(k'+1) r_j p_j$ seconds of Q_j's work has been scheduled over time interval $[0, (k'+1)p_j]$. Because Q_j is punctual, it must have generated an additional $r_j p_j$ seconds of work by time $(k'+1)p_j$. By L4, L7 and the assumption that the period of clock tick is infinitesimally small, the additional $r_j p_j$ seconds of Q_j's work receives an RC value of $(k'+2)p_j$. Using the same derivations as for the base case, but substituting $(k'+2)p_j$ for p_j (compare Figures 16a and 16b to see the similarity between the base case and the inductive case), we can prove by contradiction that the additional $r_j p_j$ seconds of work of Q_j will finish by time $(k'+2)p_j$. Hence Theorem 1 also holds for $k = k' + 1$.