CPU Service Classes for Multimedia Applications *

Hao-hua Chu, Klara Nahrstedt
University of Illinois at Urbana Champaign
h-chu3,klara@cs.uiuc.edu

Abstract

We present the design, implementation, and experimental results of our soft real time (SRT) system for multimedia applications on top of general purpose UNIX environment. The SRT system supports multiple CPU service classes for the real time processes based on their processor usage pattern including periodic constant processing time class (PCPT) and periodic variable processing time (PVPT) class. It also provides the following features: (1) reservation and processing time guarantees for the service classes, (2) overrun protection and scheduling algorithm, and (3) system-initiated adaptation strategies. The other unique feature of the SRT system is its easy portability to any operating systems with real time extensions because it is implemented purely in the user space without any modifications to the kernel. We have implemented the SRT system on the Solaris 2.6 operating system with scheduling overhead under 400*us and with good performance guarantees.*

1 Introduction

We present the design, implementation, and experimental results of our *soft real time (SRT) system*[1] for multimedia applications on top of general purpose UNIX environment and hardware environment. Current general purpose operating systems are based on Time Sharing (TS) principle which is designed to bring fairness to the multi-user and multi-process environment. However, it has been well known that multimedia applications with real time (RT) constraints do not perform well in the TS environment. On the other hand, the hard RT environment that uses specialized operating systems and hardware is considered an overkill for most of multimedia applications with soft deadlines. Hence, our solution is the SRT system that provides a *soft RT environment* with multiple service classes for different types of multimedia applications (e.g., tele-microscopy, visual tracking, 3-D animations) in the TS operating system.

In recent years, there has been an abundance of research in the area of providing real time support in the general purpose operating systems. In general, these research systems [1, 3, 4, 5, 6, 8] offer *processor time reservation* and guarantee. The RT process first enters a reservation with the system. Then a RT scheduling algorithm is used to guarantee the processor time to RT processes. They also provide *overrun protection*. An overrun is a condition when a RT process needs more processing time to complete its job[2] at a given iteration than what it has reserved. The system provides protection such that overruns from one RT process cannot cause violations to the contracts of other RT processes.

These systems work well for the class of RT processes that have constant processor usage time periodically. We define this class of RT processes as periodic constant processing time (PCPT) class. Unfortunately, many multimedia applications do not behave as nicely as PCPT. For example, their processor usage pattern may be variable rather than constant. We define a new periodic variable processing time (PVPT) class for RT processes with variable processor usage pattern.

1.1 Variable Processing Time Class

There are a few common causes that contribute to the variable processing time behavior in the multimedia applications. First, the soft RT environment consists of general purpose operating systems and hardware which do not provide any timing guarantees. For example, the unpredictable performance in the memory subsystem and the unbounded service time for system calls in the kernel all contribute to variations in the process-

*This work is supported by NSF Career Grant NSFCCR96-23867 and CISE Research Infrastructure.

[1] Our SRT system can be down-loaded with source code at our SRT homepage http://monet.cs.uiuc.edu/~h-chu3/srt.

[2] A process can release a stream of *jobs*, e.g. one job per period.

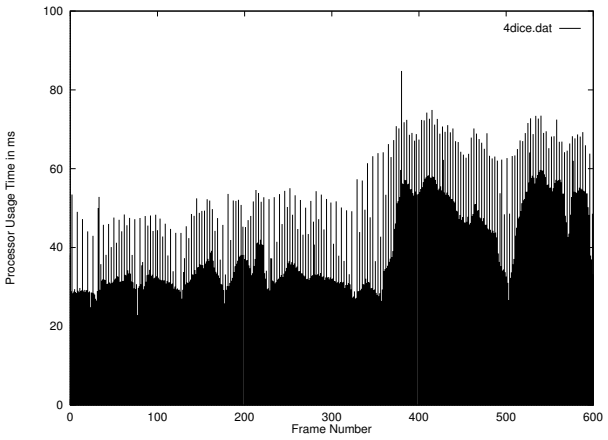

Figure 1. Frame-by-frame processing time for a MPEG decoder decoding a 352x240 stream.

ing time. Second, most multimedia applications are not programmed in such a strict timing manner as in the hard RT system because they are content dependent. For example, the amount of processor time for a software MPEG decoder can be dynamic on a frame by frame basis. The variability in processing time of a MPEG decoder can be observed in Figure 1. It is caused by a number of factors including the type of frame (I, P, B), the composition of macro-blocks in a frame, the background scene, and the amount of actions in the scene.

Systems that supports only the constant processing time class are insufficient to handle the variable processing time class of applications. For example, they may require the RT applications to make reservation close to the worst-case level. Since the worst-case rarely occurs, it leads to a waste of processor resources and low processor utilization. In Figure 1, the worst-case processing time per frame is 85ms with the average 47ms. Reserving 85ms leads to 50% waste of processor resource. It is also unnecessary to provide deadline guarantees given that users can tolerate a certain level of deadline violations. In the example of a MPEG decoder, users generally cannot detect a few milli second (ms) of delay in displaying a frame.

We can make an analogy between the CPU service classes in the processor domain and the constant/variable bit rate (rt-CBR/rt-VBR) traffic classes in the ATM network domain [2]. It is well known in the network area that constant bit rate (rt-CBR) traffic class is insufficient in handling more dynamic network applications like transmitting MPEG streams.

1.2 Processor Service Classes

We show a subset of our processor service classes provided by our SRT system in Table 1. Interested readers can visit our SRT webpage (see the footnote in the first page) for a complete list of our service classes.

Classes	Parameters	Guaranteed
$PCPT$	P (Period), PPT (Peak Processing Time)	PPT
$PVPT$	P (Period), SPT (Sustainable Processing Time), PPT (Peak Processing Time), BT, (Burst Tolerance)	SPT

Table 1. Processor service classes

The PCPT (Periodic Constant Processing Time) specification of (P, PPT) can be used to describe a process that needs at most PPT amount of processor time every period of length P. The PVPT (Periodic Variable Processing Time) specification of (P, SPT, PPT, BT) can be used to describe a process that needs on average SPT amount of processor time but no more than PPT every period of length P. In addition, the PVPT process may generate a usage burst in excess of SPT but no more than BT. A precise definition for BT is described in section 2.4.

After the service contract is established, our SRT system must check that the processes behave according to their contracts during their execution. This is called process conformance test, which is similar to the ATM traffic conformance test. If a process conforms to its contract, our system provides processor time guarantee described in Table 1.

1.3 Overrun(Burst) Partition

To accommodate the bursts from the PVPT processes, our SRT system sets aside some processor resource called *overrun partition*, and multiplex all bursts into the overrun partition. Given a small probability that all PVPT processes generate bursts simultaneously, our overrun partition can be considerable smaller than the sum of all bursts. This is the benefit of the *statistical multiplexing*. However, we cannot provide hard guarantees that bursts will always be satisfied.

1.4 Adaptation

The PVPT class alone does not solve all problems coming from variations in processing time. Consider the MPEG decoder example in Figure 1. In frames

(0-370), the decoder can establish a PVPT contract for $(SPT = 40ms, PPT = 52ms)$ which should cover most of the processing time bursts. However, there is a major scene change around frame 370 where the average processing time per frame increases from 40ms to 55ms. The contract should be adjusted accordingly to reflect the change in processor usage time pattern. For example, it should be adjusted to $(SPT = 55ms, PPT = 70ms)$ after frame 350. Our SRT system provides the *system-initiated adaptation* which can automatically adjust the parameters in the contracts for the RT processes based on their actual processor usage time.

We organize the remainder of the paper as follows: section 2 explains the architecture of our SRT system and its major components; section 3 describes our implementation; section 4 shows our experimental results; and section 5 presents our concluding remarks.

2 Design

The architecture of our SRT scheduler is shown in Figure 2. The processor resource is divided into three partitions: RT partition, overrun partition, and TS partition. The RT partition is dedicated to serving the reserved runs of RT processes, and its resource is managed by a *RT Scheduler*. The overrun partition is dedicated to serving overruns and bursts from the RT processes, and its resource is managed by an *Overrun Scheduler*. The TS partition is dedicated to serving the TS processes so that traditional TS processes do not starve because of RT processes, and its resource is managed by the underlying UNIX TS scheduler.

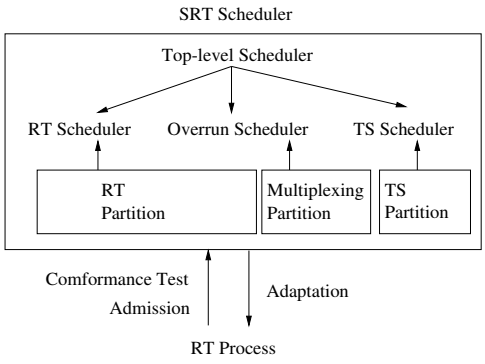

Figure 2. System Architecture.

The size of a partition is represented as a percentage of the total processor resource which is configurable by the system administrator. For the RT and TS partitions, their sizes depend on the relative weight between the RT and TS workloads. For the overrun partition, its size depends on the degree of burstiness from the RT processes. The RT, overrun, and TS partition schedulers are scheduled by a *top-level scheduler* which allocates processor time to the partitions according to their relative sizes.

2.1 Top-level Scheduler

The top-level scheduler uses a very simple credit/debit scheme to schedule the three partition schedulers. Let (C_{rt}, C_{ov}, C_{ts}) be the credits of the RT, overrun, and TS partitions. Let T_u be the length of time slice used by the top-level scheduler to schedule a partition at one time. When a partition has accumulated a credit greater than T_u units, the top-level scheduler will invoke its partition scheduler to run for one time slice T_u. It is possible that none of the partitions may have accumulated a credit of more than T_u units. For example, all partitions are initialized with 0 credit at the start of the system. Then the top-level scheduler chooses the RT partition first if $C_{rt} > 0$, or it chooses either the overrun or TS partition with a larger credit.

At the end of one time slice, the top-level scheduler debits the scheduled partition for T_u units and credits each partition for the amount proportional to their percentage values (or sizes). It is possible that the selected partition has no process to schedule. The selected partition yields its processor time back to the top-level scheduler, which allocates it to the other partitions.

2.2 RT Partition

The RT partition serves only the *reserved-runs* of the RT processes. The reserved-run is defined as the amount of guaranteed processing time specified in the contracts (see Table 1 for what is guaranteed in each class). The RT partition is managed by the RT scheduler which schedules processes so that their reserved-runs are always satisfied. In order to support such guarantee, we require that a RT process goes through a reservation phase prior to its execution phase.

A RT process starts by submitting a reservation request to the RT scheduler. Upon receiving the request, the RT Scheduler will perform an *admission control* test to determine if there is enough free processor resource in the RT partition to satisfy this request. Our RT scheduler is based on the preemptive *earliest deadline first* [7] (p-EDF) algorithm which gives the most optimal processor utilization. We assume that all RT

processes in our system are preempt-able[3]. The corresponding admission control test is as follows:

$$\sum_{i \neq s} R_i \leq \text{RT Partition}, \quad where$$

$$R = \begin{cases} PPT/P & \text{if process} \in \text{PCPT} \\ SPT/P & \text{if process} \in \text{PVPT} \end{cases}$$

After the reservation passes the admission control, it becomes a *contract*. The contract is effective immediately until the process frees it.

The RT scheduler maintains two queues: the *run queue*, and the *wait queue*. The run queue contains processes that are not overrunning their reserved processing time, and they are sorted with the earliest deadline first. The deadline of a process is the end of its current period. Processes which have completed their jobs in the current period but have yet been released for the next period are placed in the *wait queue*, which is sorted with the earliest release time first.

When the RT scheduler is invoked by the top-level scheduler, it checks the wait queue to see if any process that needs to be released for its next period. Then the RT scheduler dispatches the first process from the run queue for one time slice. At the end of one time slice, the RT scheduler can perform overrun detection and protection. If the dispatched process hasn't completed its job and it has used up all its reserved processing time, an overrun is detected. The dispatched process is removed from the run queue and inserted into one of the queues in the overrun partition. Otherwise, the dispatched process is re-inserted into the run queue.

2.3 Overrun Partition

The overrun partition serves only the *overruns* from the RT processes. We define overruns as the amount in excess of the guaranteed processing time specified in the contracts (see Table 1). The overrun partition acts as a shared resource where all the bursts and overruns are multiplexed into. The overrun partition is managed by the overrun scheduler. Unlike the RT scheduler, the overrun scheduler makes no guarantee on the amount of processor time that each overrun process receives, or that each overrun will be scheduled in time to meet its deadline.

The overrun scheduler distinguishes between *conforming overruns* and *non-conforming overruns*. Conforming overruns are defined as bursts from the PVPT processes that exceed their SPT, but still conform to their contracts. Non-conforming overruns are defined as bursts from any processes that do not conform to their contracts. We will describe the conformance tests in section 2.4. The overrun scheduler maintains three queues: *conforming queue* for conforming overruns, *non-conforming queue* for non-conforming overruns, and *permanent non-conforming queue* for non-conforming overruns occurring frequently over a long period of time.

When the overrun scheduler is invoked by the top-level scheduler, it schedules the conforming overruns first. When there are no conforming overruns, it then schedules the non-conforming overruns and last the permanent non-conforming overruns. Within each overrun queue, the overrun scheduler uses a *round robin* algorithm which gives each overrun an equal share of the overrun partition. The emphasis is on *fairness*. All overrun queues are FIFO (first in first out). The overrun scheduler inserts a new overrun at the end of the queue regardless of its deadline and dispatches from the head of the queue.

At the end of every time slice (during the interrupt point), the overrun queues need to be checked if any periodic overrun process has passed its deadline. This is called a *deadline miss*. A flag is set to notify the process of a deadline miss.

2.4 Process Conformance Test

The process conformance test checks if the process behaves according to its contract. The system-specific burst tolerance ($SSBT$) is defined as the variation caused by our scheduling overhead, the underlying general purpose operating system, and hardware.

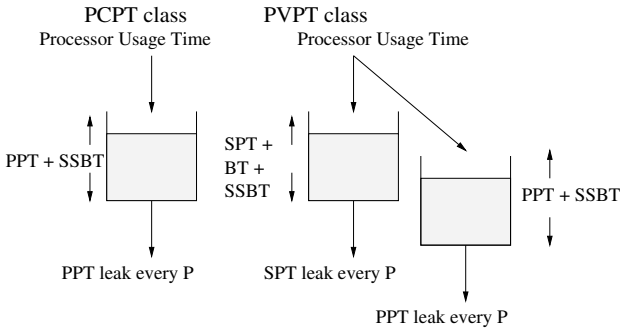

Figure 3. Process conformance test for the service classes.

The conformance test for the PCPT class can be represented by a leaky bucket in Figure 2.4. The bucket

[3]This is generally true for processes in the general purpose operating systems.

has depth $PPT+SSBT$. The amount equivalent to the processor usage time is poured into the bucket. Every period P, the bucket is drained for the amount PPT. The process conforms to its contract when its bucket does not overflow. The conformance test for the PVPT class can be represented by two leaky buckets. The left bucket has a depth $SPT + BT + SSBT$ and the right one has depth $PPT + SSBT$. The amount equivalent to the processor usage time is poured into both buckets simultaneously. Every period P, the left bucket is drained for the amount SPT, and the right one PPT. The process conforms to its contract when both buckets do not overflow.

The conformance test is invoked when a process is dispatched for a time slice, or when a process encounters the end of its periodic deadline.

2.5 Adaptation

System-initiated adaptation can automatically adjust the parameters in the contracts for RT processes based on the monitored processor usage time. A RT application can choose from the following two adaptation strategies. We denote X as the guaranteed parameter (PPT for a PCPT process and SPT for a PVPT process) in the contract.

The exponential average strategy is based on the following formula:

$$X_i = (1 - \alpha) * X_{i-1} + \alpha * X_i$$

The RT application selects the parameter $0 < \alpha < 1$ that represents the relative weight between the current X_i value and the previous X_{i-1} value in determining the new X_i value. The default value for α is 0.5. The RT application also selects the window size (ws) which represents the number of past iterations from which the new X value is calculated from. For example, a PCPT process with $ws = 10$ means that the peak processing time is calculated as the maximum processing time over the last 10 iterations. The window size controls the frequency of adjustment on X. Since each adjustment requires a re-negotiation of processor resources, frequent adjustments can be undesirable.

The statistical strategy requires two parameters: frequency of non-conforming overruns(f) and window size (ws). It will adjust X to a level where no more than $(f * ws)$ number of non-conforming overruns can occur within the ws number of iterations. For example of $(f = 0.2, ws = 5)$ and a process has its most recent 5 processing usage history as $(53ms, 50ms, 30ms, 40ms, 55ms)$, statistical adaptation will adjust $PPT = 50ms$ so that only 2 non-conforming overruns ($> 50ms$) occur.

3 Implementation

Another novel feature of our SRT system is that it is implemented purely in the user space without any modifications into the kernel. The implementation requires that the underlying operating system provides the following two services (1) a real time interval timer (e.g. *setitimer*()), (2) fixed priorities (e.g. *priocntl*()). These two services are available in almost all UNIX systems under the POSIX.4 real time extension.

The essence in the implementation of the our server is in the *dispatch mechanism*. The SRT server process must have root privilege so that it can manipulate the fixed RT priorities and it can change the priority of RT client processes. The server process runs at the highest possible global(and fixed) priority. At the system startup time, the server sets the real time interval timer equal to the time slice T_u, which means that the timer will wake up (signal) the server process every T_u time. After a RT process *pid* is selected to be dispatched, the server promotes $pid's$ priority from lowest to the 2nd highest global (and fixed) priority. Then the server sleeps, and *pid* is scheduled by the underlying kernel given that it is the runnable process with the highest priority. At the end of one time slice, the interval timer wakes up the server, which also preempts *pid*. The server stops(suspends) *pid* and lowers $pid's$ priority from the 2nd highest priority to the lowest priority. This dispatch cycle repeats.

The server allows the TS processes to run by not promoting any RT processes to the dispatch priority. As a result, the kernel will schedule the TS processes according to its TS scheduler.

We have implemented our SRT system on a single processor Sun Ultra Sparc machine running Solaris 2.6 operating system. The *dispatch latency*, which the amount from the end of previous dispatched process to the start of the next dispatched process, is measured to be consistently less than 400us.

4 Experiment

The experiment consists of the following mixture of RT and TS processes running concurrently. TS-1: a compilation process. TS-2 and TS-3: a computation intensive program. RT-4: a PVPT class MPEG decoder plays the 352x240 stream in Figure 1 at 5 frames per second. The PVPT parameters are $(P = 200ms, SPT = 40ms, PPT = 52ms, BT = 15ms)$ with statistical adaptation strategy $(f = 0.2, ws = 10)$. RT-5: a PVPT class MPEG decoder plays the 192x144 stream at 10 frames per second. The PVPT parameters are $(P = 100ms, SPT = 15ms, PPT = 25ms, BT =$

$10ms$). RT-6: a PCPT class monitor program takes a sample every 50ms. The PCPT parameters are ($P = 50ms, PPT = 500us$) without any adaptation. RT-7: a PCPT class misbehaving program uses as much processor time as possible. Its PCPT parameters are ($P = 500ms, PPT = 10ms$). This is to show our overrun protection.

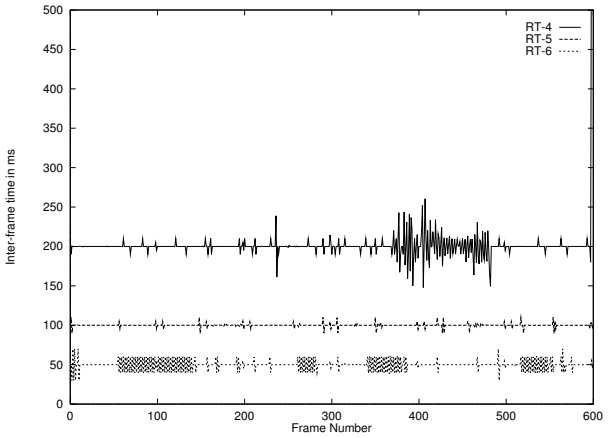

Figure 4. The inter-frame time for the three behaving RT processes.

Figure 4 plots the inter-frame time for the three behaving RT processes RT-4, RT-5, and RT-6. There is no deadline miss for them, and the jitter for the two MPEG decoders are all kept within one frame time. This shows that the misbehaving RT-7 process does not affect the behaving RT processes. Figure 5 shows how the statistical adaptation strategy adjusts the SPT parameter for the RT-4 MPEG decoder process. The SPT value is adjusted from the initial 40ms to 56ms when the processor usage time increases after frame 370.

5 Conclusion

Our SRT system stands out among the related system by introducing various real time service classes in the processor domain. Our system also provides unique features like exponential average and statistical adaptation strategies, and overrun scheduling. Our system can achieve low overhead and provide good guarantees to RT processes using a user-space implementation and without any kernel modifications. We have also shown through experimental result that our SRT system can deliver good guarantees to the RT processes.

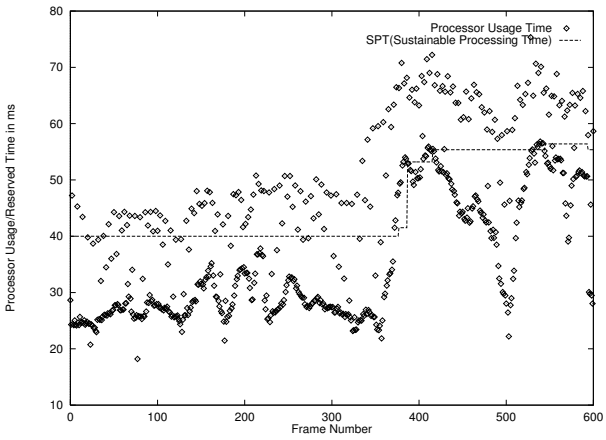

Figure 5. Statistical adaptation strategy adjusts SPT (Sustainable Processing Time) shown as the line for RT-4. The dotted line represents the actual usage.

References

[1] H. Chu and K. Nahrstedt. A soft real time scheduling server in unix operating system. In *European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, Darmstadt, Germany, September 1997.

[2] A. F. T. Committee. Traffic management specification version 4.0. *http://www.atmforum.com/atmforum/specs/approved.html*, April 1996.

[3] Z. Deng, J.-S. Liu, and J. Sun. Dynamic scheduling of hard real-time applications in open system environment. In *Real-Time Systems Symposium*, Washington, DC, December 1996.

[4] P. Goyal, X. Guo, and H. Vin. A hierarchical cpu scheduler for multimedia operating system. In *Second Usenix Symposium on Operating System Design and Implementation*, Seattle, WA, October 1996.

[5] M. B. Jones, D. Rosu, and M. Rosu. Cpu reservation and time constraints: Efficient, predictable scheduling of independent activities. In *16th ACM Symposium on Operating Systems Principles (SOSP '97)*, St. Malo, France, October 1997.

[6] C. Lee, R. Rajkumar, and C. Mercer. Experience with processor reservation and dynamic qos in real-time mach. In *Multimedia Japan*, March 1996.

[7] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *JACM*, pages 20(1):46–61, 1973.

[8] D. Yau and S. Lam. Adaptive rate-controlled scheduling for multimedia applications. In *ACM Multimedia Conference*, Boston, MA, November 1996.