A Soft Real Time Scheduling Server in UNIX Operating System

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

Abstract

We present a soft real-time CPU server for Continuous Media processing in the UNIX environment. The server is a daemon process from which applications can request and acquire soft real-time QoS (Quality of Service) Guarantees. Our server architecture addresses in addition to other multimedia CPU scheduling extensions properties such as fairness, QoS brokerage and enforcement, and security. Overall it provides (1) protection among real-time(RT) processes, (2) fairness among RT and non-RT processes, (3) rate monotonic scheduling, and (4) a fix to the UNIX security problem. We have implemented our soft real-time CPU server in the SUN Solaris 2.5 Operating System, and we have shown through experiments that our soft RT server provides predictable QoS for continuous media applications.

Keywords : Soft Real-Time Scheduling, Quality of Service, Unix, Continuous Media Processing.

I. Introduction

Continuous media such as Video and Audio are becoming widely-used in computer applications nowadays. To preserve their time-sensitive (temporal) behavior, multimedia applications require that the underlying systems provide soft real time Quality of Service (QoS) guarantees.[1] However, in the current UNIX multi-user, multi-process, and time-sharing (TS) environment, these applications do not perform very well when they are scheduled concurrently with the traditional non-RT applications such as text editors, compilers, web browsers, or computation-intensive tasks. These soft real-time (RT) applications also do not perform very well when other RT applications are scheduled concurrently. Part of the problem is due to untimely scheduling of processes rather than insufficient CPU capacity. This paper addresses this problem and presents an extension to the current UNIX scheduler - a soft real time server - to schedule soft RT applications in the more predictable fashion.

One possible solution to serve RT application in UNIX environment is to dedicate the entire system to serve only one RT application. This involves blocking services to all other RT or non-RT applications and users. This solution avoids the potential scheduling problem, and it also defeats the UNIX environment goals of supporting multi-user, multi-process and time-sharing properties. Therefore, this solution is not feasible in the UNIX environment.

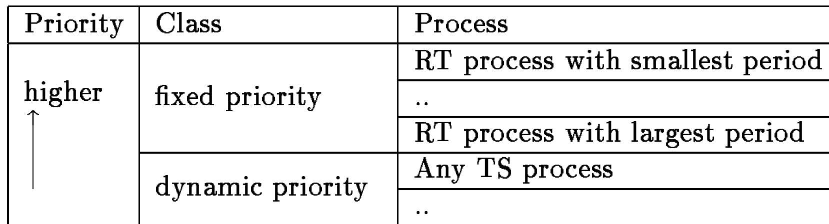
Another more feasible solution is the current RT extension to UNIX. The UNIX/POSIX.4 real time extension provides fixed priorities to real time applications. The priority scheduling rule dictates that higher priority

*This work is supported by National Science Foundation Career Grant, under contract: NSFCCR96-23867 and CISE Instructional Equipment Grant.

[1] Soft real time guarantee means that there could exist a time when guarantees are violated, e.g. due to page faults, but this violation is bounded.

TABLE I

UNIX/POSIX.4 RT/TS PRIORITY CLASS WITH THE RATE MONOTONIC EMULATION



Priority	Class	Process
higher ↑	fixed priority	RT process with smallest period
		..
		RT process with largest period
	dynamic priority	Any TS process
		..

processes are scheduled before the lower ones in a preemptive fashion. RT processes are assigned higher fixed priorities, whereas non-RT processes are assigned lower dynamic priorities. As a result, the RT processes are served before non-RT processes, and higher priority RT processes are served before lower priority RT processes. This fixed priority mechanism provides a convenient way to implement the rate monotonic (RM) algorithm because the ordering of priorities between the RT processes depends on the ordering of the process rates (the length of their periods) as shown in the Table I. Under this RM schedule[3], the RT processes with smaller periods are executed first, followed by RT processes with larger periods, and then non-RT processes. Prior to the start of any RT process, the schedulability test is performed by checking its total CPU demand so that including this new process, the CPU resource allocation will not exceed the CPU capacity. But this schedule has many problems described below.

- Priority should represent the importance of a process rather than whether this process is RT or non-RT. For example, is user A playing a song (a RT application) more important than user B writing a term paper (a non-RT application)? RT and non-RT applications must share the CPU time fairly. It is also unreasonable to assign priority for RT applications based on the length of their periods. For example, is a video player running at 30 frames per second more important than another video player running at 20 frames per second? This is called the *fairness* problem.
- It offers no protection between applications. There isn't any mechanism to enforce deadline and to guard against overrun and monopoly of CPU by a faulty or greedy RT process at high priority. Frequent overruns from a high priority process can cause starvation to other processes at lower priority. A run-away RT process at very high priority can even block most of the system processes and lock up the entire system. This is called the *enforcement* problem.
- It requires root privilege to use the fixed priority. From a UNIX security viewpoint, it is impossible to give every user root privilege to run RT applications. This is called the *security* problem.

In this paper, we describe a soft RT scheduling server in UNIX OS environment that will address the *fairness, enforcement, and security* problems mentioned above without compromising the flexibility and efficiency of the UNIX Operating System. The approach of our RT server is general, it requires no modification to the

kernel and it can be easily ported to almost any derivations of UNIX systems. Our RT server is intended for scheduling of soft RT applications only, such as multimedia applications that can tolerate some variations in temporal quality of service.

The paper is organized as follows: Section 2 describes the related works; Section 3 explains the scheduling server architecture; Section 4 describes the implementation on SUN Solaris; Section 5 shows the experimental results; Section 6 presents the concluding remarks.

II. Related Works

The area of accommodating scheduling of soft RT applications on the current UNIX platforms was addressed by several groups. Goyal, Guo, and Vin [2] implemented the *Hierarchical CPU Scheduler* in the SUN Solaris 2.4. The CPU resource is partitioned into hierarchical classes, such as Real-time and Best-Effort classes, in a tree-like structure. A class can further partition its resource into subclasses. Each class can designate a suitable scheduler to meet the objective of its processes(leaf nodes). Protection between classes is achieved by the *Start-time Fair Queuing* (SFQ) algorithm which is a modification of the Weighted Fair Queuing Algorithm. SFQ is a fair scheduler that schedules all the intermediate classes and subclasses according to their partitioned resources. The major disadvantage of this approach is that their implementation requires modifications to the Solaris kernel scheduler. Fair sharing also does not translate directly into applications QoS guarantees that require a specific amount of CPU allocation and a constant periodicity. Furthermore, the scheduling overhead can be expected to rise proportionally with increasing depth and breath of the hierarchical trees.

Mercer, Savage, and Tokuda [4] implemented the *Processor Capacity Reserves* abstraction for the RT-threads in the RT Mach Operating System. A recent version [1] supports dynamic Quality adjustment policy. A new thread must first request its CPU QoS in the form of *period*, and *requested CPU usage in percentage* during the reservation phase. Once it is accepted, a *reserve* of CPU processing time is setup and it is binded to this new thread. Any computation time done on behalf of this process, including all service time from various system threads, is charged to this reserve. The reserve is replenished periodically by its requested CPU usage time. This concept is similar to the *Token Bucket*. The accurate accounting of system service time is a superior but costly feature. It requires non-trivial modifications and computation overhead inside the UNIX kernel to support this abstraction, such as keeping track of the reserves database, and passing the client process's reserve to and between system threads.

Yau and Lam [13] implemented the Adaptive Rate-Controlled Scheduling, which is a modification of the *Virtual Clock* Algorithm. Each process specifies a reserve rate and a period for its admission control phase. During its execution phase, the reserve rate is adjusted upward or downward to match its actual usage rate in a gradual fashion. It is called *rate adaptation*.

Gopalakkrishnan [12] implemented the *Real Time Upcall (RTU)* on the NetBSD UNIX to support periodic tasks. Each RTU is similar to a process, it contains an event handler that registers to the kernel its execution

TABLE II
URSCHED PRIORITY STRUCTURE

	Priority	Process
RT class	highest	URsched scheduler
	2nd highest	Running RT process
	..	Not used
TS class	any	Any TS processes
RT Class	lowest	Waiting RT processes

time and period. The kernel dispatcher is modified to schedule the RTUs using the Rate Monotonic algorithm. In order to increase the predictability and efficiency, the kernel dispatcher disallow preemptions during the middle of the RTU execution, called *delay preemption* and no asynchronous preemption.

Kamada, Yuhara, and Ono [7] implemented the *User-level RT Scheduler* (URsched) in the SUN Solaris 2.4. The URsched approach is based on the POSIX.4 fixed priority extension and its priority scheduling rule. The scheduler runs at the highest possible fixed-priority, the RT process waits its scheduling turn at the lowest possible fixed-priority (called the waiting priority), and the active RT process runs at the 2nd highest fixed-priority (called running priority). The priority structure is shown in Table II. The scheduler wakes up periodically to dispatch the RT processes by moving them between the waiting and the running priority; during the other time, it just sleeps. When scheduler sleeps, the RT process at the running priority executes. When no RT processes exists, the non-RT processes with dynamic priorities execute using the fair time sharing scheduler of UNIX. This provides a simple mechanism to do RT scheduling in UNIX. It also has many desirable properties which the previous two approaches do not provide:

- It requires no modification to the existing UNIX/POSIX.4 kernels. The scheduling process can be implemented as an user-level application.
- It has very low computation overhead.
- It provides the flexibility to implement any scheduling algorithms in the scheduler, e.g. rate monotonic, earliest deadline, or even the Hierarchical CPU Scheduler.

In this paper, we will apply lessons learned from the previous approaches and extend the existing solutions to build a higher level soft RT scheduling server. In addition to the properties of the URsched, we provide the following functionalities:

- New Design of the *Soft RT Server Architecture*.
- Implementation of *Rate Monotonic Scheduling*.
- Provision of *Protection* among real-time processes.
- Fixing the *Security problem* in UNIX Operating System.

III. Server Architecture Design

Our server architecture contains three major components–the **broker**, the **dispatcher**, and the **dispatch table** as shown in Figure 1. Each component is described in details in the following sections.

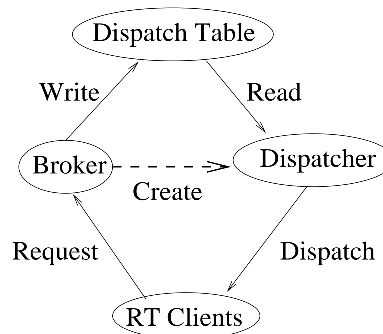


Fig. 1. The Soft RT server architecture

A. Broker

The broker receives requests from client RT processes. It performs the admission control test to determine whether the new client process can be scheduled. If it is schedulable, the broker will put the RT process into the waiting RT process pool by changing it to the waiting priority. The broker also computes a new schedule based on a desirable scheduling algorithm, the new schedule is written to the *dispatch table*.

The broker process is a root daemon process running at a normal dynamic priority. It can be started at the system boot time, like any other network and file system daemons. It will wake up when the new client request arrives. The broker needs to be a root process so that it can change processes into the fixed priority. The broker process does not perform the actual dispatching of the RT processes, instead it will fork a separate real-time *dispatcher* process. The reason is that the admission and schedulability test in the broker may have variable computation time, hence it may affect the timing of dispatching. The admission and schedulability test do not need to be done in real time; as a result, the broker runs at a dynamic priority. The separation of RT dispatching in the *dispatcher* process and the non-RT *schedulability test* in the *broker* process is an essential feature that allows both *dispatcher* and *broker* to do on-line computation without compromising the precision of RT processes dispatching.

The client RT processes must start their processing at the non-RT dynamic priority level. The broker and dispatcher will change them into the fixed RT priority when they are accepted and dispatched. This is an improvement over the current UNIX environment, our scheme allows any user to run processes at the fixed priority in a fair and secure manner.

TABLE III
SAMPLE DISPATCH TABLE, REAL TIME

Slot Number	Time	Process PID
0	$0 - 10ms$	721
1	$10 - 20ms$	773 774 775
2	$20 - 30ms$	721
3	$30 - 40ms$	free
4	$40 - 50ms$	721
5	$50 - 60ms$	773 774 775
6	$60 - 70ms$	721
7	$70 - 80ms$	free

B. Dispatch Table

The *dispatch table* is a shared memory object where the *broker* writes the computed schedule to and the *dispatcher* reads from in order to know how to dispatch RT processes. It is locked inside memory for efficient read and write. The dispatch table contains a repeatable *frame* of slots, each slot corresponds to a time slice of CPU time. Each slot can be assigned to a RT process pid, a group of cooperating RT process pids, or be free which means yielding the control to the UNIX TS scheduler to schedule any non-RT processes. Consider the example in Table III. The repeatable *frame* is $80ms$, and it contains 8 time slots of $10ms$ each. The sample dispatch table is a result of a RM schedule with the pid 721 at *period*=$20ms$, *execution time*=$10ms$, and pid 773/774/775 at *period*=$40ms$, *execution time*=$10ms$. There are 2 free slots, which means $20ms$ out of every $80ms$ of CPU is allocated to the non-RT processes.

The minimum number of free slots is maintained by the broker to provide a fair share of CPU time to the non-RT processes. In the Table III, 25% ($20ms$ out of $80ms$) of the CPU is guaranteed to the non-RT processes. The site administrator can adjust the non-RT percentage value to be what is considered fair. For example, if the computer is used heavily for RT applications, the non-RT percentage can be set to a small number, and vice versa.

C. Dispatcher

The *dispatcher* is a process running at the highest possible fixed priority. The dispatcher process is created by the broker and it is killed when there are no RT processes to be scheduled in the system. When there are only non-RT processes running, the system has no processing overhead associated with the RT server.

The dispatcher contains the shared memory dispatch table and a pointer to the next dispatching slot. At the beginning of the next dispatch slot, a periodic RT timer signals the dispatcher to schedule the next RT process. The length of time to switch from the end of one slot to the start of the next one is called the *dispatch latency*. The dispatch latency is the scheduling overhead which should be kept at a minimal value.

The dispatching process is similar to the URsched approach. Consider the dispatch table in Table III at time $50ms$. The dispatcher is moving from slot 4 to slot 5, the following steps are taken.

- The periodic RT timer wakes up the dispatcher process, and the process 721 is preempted (1 context switch).
- The dispatcher changes the process 721 to the waiting priority and processes 773/774/775 to the running RT priority (4 system calls to set priority).
- The dispatcher puts itself to sleep, and one of the processes 773/774/775 are scheduled (1 context switch).

The program code segment that corresponds to the above steps is executed repeatedly, and is locked into memory to avoid costly page faults. The dispatch latency can be bounded by the time to do 2 context switches and (the maximum number of processes in any 2 adjacent slots) set-priority system calls.

Unless the processes are in their critical sections, dispatcher will not allow process to overrun. The execution time of the RT processes execution time is protected from each other.

D. RT clients

Our client's system QoS request uses the simple RT Mach's form of QoS specification: *period=p, CPU usage in percentage=u*. For example, the specification ($p = 10ms, u = 20\%$) means that $2ms$ out of every $10ms$ are reserved for this RT process. There are restrictions to the size of the period. The hardware restricts the size to be greater than the software clock resolution. Smaller period also leads to smaller time slice, which results in higher number of context switchings and inefficient CPU utilization.

The users can compute the client's required period (e.g. 40 frames per second video player has a period of $25ms$), but the computation of the CPU usage percentage is very difficult. The UNIX command *rusage* will not give an accurate system time because some of the system threads execution times may not be accounted, as described in the Process Reserves Model in the RT Mach [4]. The user can get a good estimation by a few trial-and-error runs of the client programs.

The execution time also depends on the state of memory contention and the resulting number of page faults. We are currently designing a *memory broker* where the RT process can reserve memory prior to their execution.

The client RT process may block while making a system call. During this blocking time, the client is not executing. But since the kernel is serving the system call for the client, the blocking time should be considered as a part of client's execution time (as in the case of Process Reserves Model).

IV. Implementation

A. SUN Solaris 2.5 Implementation

We have implemented our server architecture on a single processor Sun Sparc 10 running Solaris 2.5 OS. The Solaris OS has a default global priority range (0-159), 0 the least importance. There are 3 priority classes:RT

class, System class, and TS class. The RT class contains fixed priority range (0-59), which maps to the global priority range (100-159). The dispatcher's priority is 59, the running priority is 58, and the waiting priority is 0. The waiting priority 0 needs to be mapped to the lowest global priority 0, and it must be lower than any TS priorities. This can be done by compiling a new RT priority table *RT_DPTBL* inside the kernel.

The changing priority is done by using the *priocntl()* system call. Its average cost is measured as $175\mu s$. The average dispatch latency (2 context switch + 2 *priocntl()* is measured as $1ms$. The interval timer is implemented using *setitimer()* with the maximum resolution of $10ms$ and it is the smallest time slot size. The overhead is 10%, which is acceptable.

The broker implements a rate monotonic(RM) scheduling algorithm. In our future work, we will implement the earliest deadline first (EDF) algorithm.

V. Experimental Results

We have tested our soft RT server with two experiments which run concurrently multimedia application(s) and non-RT applications. The metric to evaluate the performance of our CPU server is the **jitter** of the continuous media in multimedia application(s).

A. Experiment 1

The first experiment consists of the mixture of the following 4 popular applications running concurrently:

- The Berkeley mpeg_play (version 2.3) plays the TV cartoon Simpsons mpeg file at 10 frames per second.
- The gcc compiler compiles the Berkeley mpeg_play code.
- Our program computes the *sin* and *cos* table using the infinite series formula.
- The Latex program formats this paper.

The graphs in Figure 2 show the measurement of **frame jitter** on the mpeg_play under the above specified load. The left graph shows the result of the normal TS UNIX scheduler without our server. The right graph shows the result of our server with 60% CPU reserved every $50ms$ to the mpeg_play. Using the UNIX TS scheduling, noticeable jitter over $100ms$ (equivalent to 1 frame time) occurs frequently–61 times out of the 650 frames (65 seconds). The largest jitter is about $350ms$ (over 3 frames time), which is clearly unacceptable. Using our server, noticeable jitter over $100ms$ occurs only once.

The Table IV shows the elapsed time of the other three applications–compute, compile, and Latex. Using our server, their execution time increases by no more than 20%, which is acceptable for non-RT applications.

B. Experiment 2

The second experiment adds a second mpeg_play that plays the same mpeg file at 5 frames per second. The graphs in Figure 3 show the measurement of jitter on two mpeg_play applications. The top graphs show the results of the 10 frames per second mpeg_play and the bottom graphs show the results for the 5 frames per second. The left graphs show the results of the normal TS UNIX without our server. The right graphs show

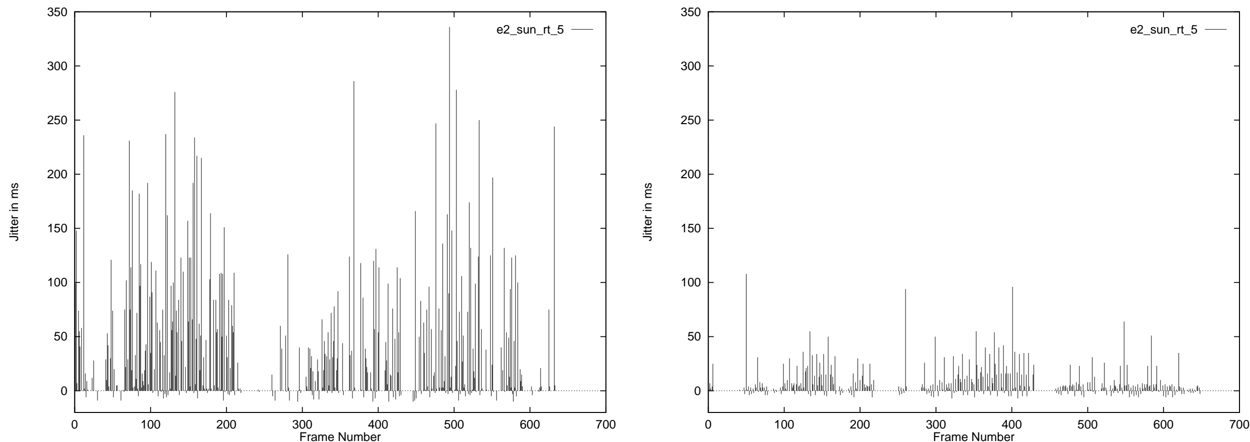


Fig. 2. Frame Jitter Measurement for mpeg_play at 10 frames per second on the SUN Sparc 10 Solaris 2.5 platform. The left graph shows the result for the TS UNIX Scheduler. The right graph shows the result of our RT scheduling server with 60% CPU reserved every $50ms$.

TABLE IV
Elapsed Time of the compute, compile and Latex applications.

	Compute	Compile	Latex
TS UNIX	$75.11s$	$141.36s$	$28.93s$
Server	$90.90s$	$160.33s$	$33.24s$

the results of our server with 60% CPU reserved every $50ms$ to the first mpeg_play, and 30% CPU reserved every $50ms$ to the second mpeg_play. Using the UNIX TS scheduling, noticeable jitter over $100ms$ occurs frequently–153 times out of 650 frames (65 seconds) for the first mpeg_play, and 10 times out of 325 frames for the second mpeg_play programs. The largest jitter is about $400ms$ (4 frames time), which is unacceptable. Using our server, noticeable jitter over $100ms$ occurs 0 and 1 time for the first and second mpeg_play programs.

The Table V shows the elapsed time of the other 3 applications. Using our server, their average execution time increases by 50%, which is still acceptable for non-RT applications.

VI. Conclusion and Future Work

We have shown through experiments that our soft RT server provides predictable QoS for Continuous Media applications in the UNIX environment. It addresses *fairness* by maintaining a minimum amount of CPU time

TABLE V
Elapsed Time of the compute, compile and Latex applications.

	Compute	Compile	Latex
TS UNIX	$102.84s$	$283.10s$	$33.70s$
Server	$132.86s$	$293.04s$	$69.53s$

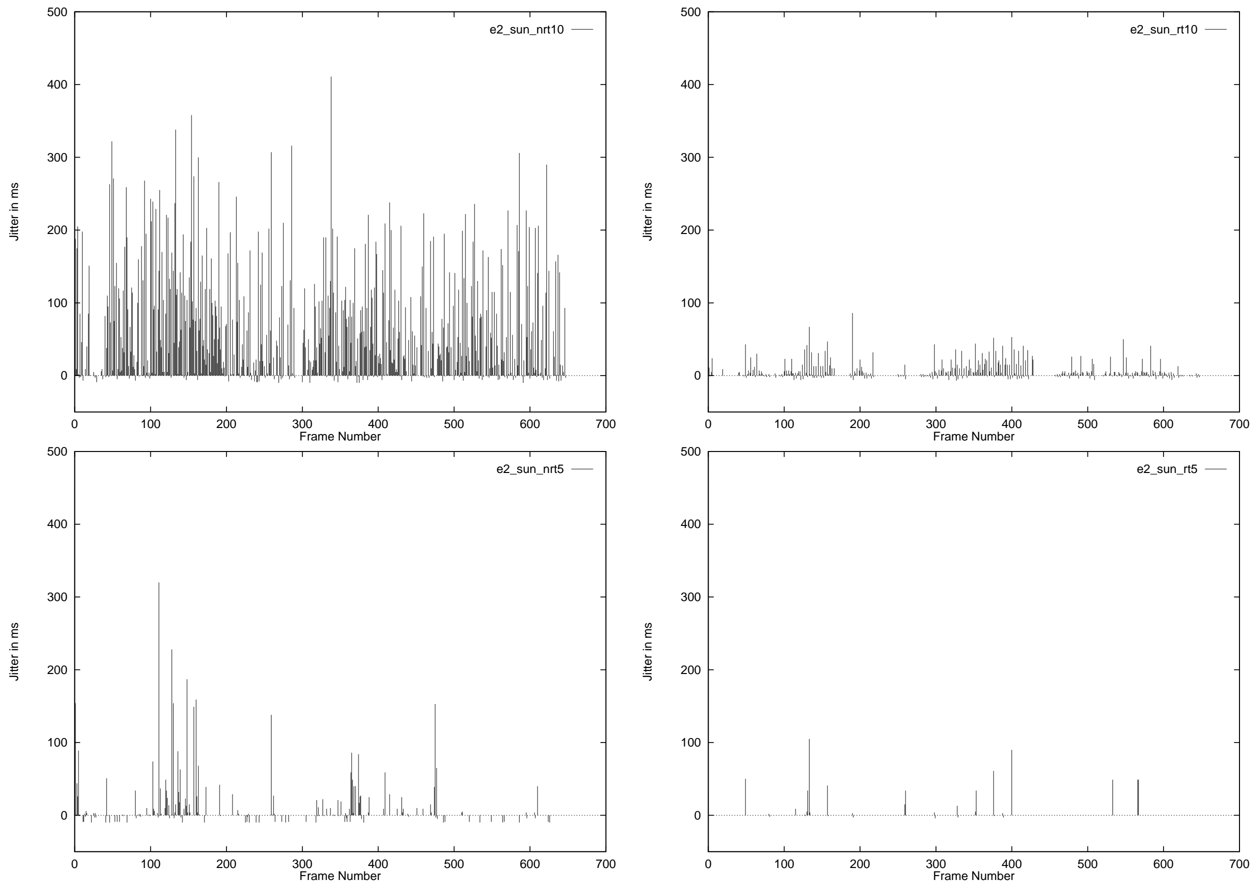


Fig. 3. Jitter Measurement for the two mpeg_plays at 10fps(top) and 5fps(bottom) on the SUN Sparc 10 Solaris 2.5 platform. The left graphs shows TS UNIX. The right graphs shows our server with 10fps(top) 60% CPU reserved every $50ms$, and 5fps(bottom) 30% CPU reserved every $50ms$.

for the non-RT processes, it offers *protection* by disallowing RT process to overrun their time slots, and it provides *security* through the broker interface. In our future work, we will incorporate our soft RT server in our general *Resource Broker Architecture* that provides better QoS for soft real-time applications. We list three areas which our *Resource Broker Architecture* will address:

- Memory Management.
- Adaptive Resource Management and Dynamic QoS Request
- QoS Translation Service

A. Memory Management

Our current resource management controls only the CPU, but this is not sufficient. Memory management is also important. It is well known that the performance of the virtual memory system has a great impact on the process performance, or its execution time. For example, a page fault can take up to $10ms$ to service on a SUN sparc 10 workstation, and it can disrupt the timing of the real-time process. Page faults can occur frequently when processes are competing for physical memory. Unfortunately most of the Continuous Media applications

that deal with video and audio are memory intensive even when compression is used. The frequency of the page fault must be minimized in order to provide predictable CPU execution time. As a result, the system must provide a memory management scheme that allows real-time processes to reserve and to lock certain critical sections of their program and data segments into physical memory.

B. Adaptive Resource Management and Dynamic QoS Request

The second not well understood issue is the adaptive resource management and dynamic QoS request. Most of the current protocols rely on the static resource allocation and static QoS request. These protocols require that a real-time process submits its QoS request during the reservation phase. Then admission control is performed to check for resource availability; if it is successful, resource is allocated for the entire duration of the real-time process regardless of its actual resource consumption and the changing load on the resources. For example, a video playback application may initially request the most desirable QoS of 30 fps (frames per second), but when resource load is high, it can tolerate a degradation to 20 fps.

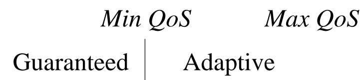


Fig. 4. Dynamic QoS Request

Our designed protocol supports dynamic QoS request in the form of a range $\{MinQoS, MaxQoS\}$ as shown in Figure 4. The $MinQoS$ is the guaranteed portion to the real-time process, it should be the minimum level of resource that real-time process needs in order to perform its task. The adaptive portion, up to $MaxQoS$ is dynamically adjusted by the resource management depending on its current resource consumption and the overall availability of the resources. This is similar to the Imprecise Computation model [14].

C. QoS Translation Service

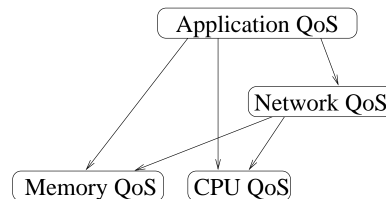


Fig. 5. QoS dependency graph for QoS Translation service

The third issue is the complexity of the QoS specification. Most protocols require the specification of the QoS parameters at the resource level, e.g. CPU time in period and milliseconds, memory usage in bytes, and network bandwidth in Mbits per second. However, it is difficult to translate the higher level application QoS parameter into these lower level resource QoS parameters. Take the example of a network mpeg video player,

where the *application QoS* is specified in frame-rate, e.g. 10 frames per second(fps). How much memory or CPU resources does 10 fps translate to? As shown in Figure 5, the *application QoS* needs to be translated into *network QoS* for its network processing task, and *memory and CPU QoS* for its decode and display tasks. Furthermore, the Network Protocol Stack Processing is a part of the network task, so the *network QoS* needs to be translated into *memory and CPU QoS*. This forms the *QoS dependency graph* which the QoS translation service needs to take into an account and resolve the resource mappings.

References

[1] Chen Lee, Ragunathan Rajkumar, and Cliff Mercer. "Experience with Processor Reservation and Dynamic QOS in Real-Time Mach". *Multimedia Japan*, 1996.

[2] Pawan Goyal, Xingang Guo, and Harrick Vin. "A Hierarchical CPU Scheduler for Multimedia Operating System". *The proceedings of Second Usenix Symposium on Operating System Design and Implementation.*

[3] Bill O. Gallmeister. "Programming for the Real World: POSIX.4". O'Reilly & Associates, INC. 1995.

[4] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. "Processor Capacity Reserves: Operating System Support for Multimedia Applications". *IEEE International Conference on Multimedia Computing and Systems.* May 1994.

[5] Klara Nahrstedt. "Time-variant QoS Management". NSF Career Proposal. 1995.

[6] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications". *Fourth International Workshop on Network and Operating System Support for Digital Audio and Video.* November 1993.

[7] Jun Kamada, Masanobu Yuhara, Etsuo Ono. "User-level Realtime Scheduler Exploiting Kernel-level Fixed Priority Scheduler". *Multimedia Japan*, March 1996.

[8] Sandeep Khana, Michael Sebree, and John Zolnowsky. "Realtime Scheduling in SunOS 5.0". *USENIX Winter 1992 Technical Conference.*

[9] "The Real-time Frame Scheduler in IRIX Programmer's Guide". Silicon Graphics, Inc. 1994.

[10] "System Interfaces Guide". Sun Microsystems Inc. November 1995.

[11] "SunOS Multi-thread Architecture". Sun Microsystems Inc. August 1995.

[12] Gopalakrishnan, R. "Efficient Quality of Service Support Within Endsystems for High Speed Multimedia Networking". PhD Thesis, Washington University. December 96.

[13] David K.Y. Yau and Simon S. Lam. "Adaptive Rate-Controlled Scheduling for Multimedia Applications". ACM Multimedia Conference '96, Boston, MA, Nov 1996.

[14] Jane Liu, Kwei-Jay Lin, and Swaminathan Natarajan. "Scheduling real-time, periodic jobs using imprecise results". In Proceedings, Real-Time Systems Symposium, pages 252-260, San Jose, California, December 1987.