Performance Evaluation of CPU Isolation Mechanisms in a Multimedia OS Kernel

David K.Y. Yau
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
yau@cs.purdue.edu

Abstract

To allow continuous media applications fine-grained control over their CPU allocations, and to protect these allocations from each other, thread priorities must have quality-of-service (QoS) interpretation. To this end, we present a CPU scheduler based on the well-defined resource specification of service curve. Service curve is distinguished from the traditional notion of rate by its ability to flexibly decouple delay and rate performance. Apart from how we compute thread priorities, predictable performance is hard to achieve also because threads can interact with each other and contend for synchronization resources. Such interactions can contribute to various forms of priority inversion. We discuss a new approach of dynamic priority inheritance in our CPU scheduler that solves priority inversion due to lock contention. To solve priority inversion arising from incompatible client/server resource specifications, we employ a train abstraction that allows a thread of control to visit multiple protection domains while carrying its resource and scheduling state intact. Train has been applied to real applications like a Solaris X window server. Finally, we present a mechanism for Internet flow specifications to reserve CPU time for network receive interrupt processing. We demonstrate an experimental system in which the combined techniques provide effective CPU isolation under various conditions of lock contention, client/server programming, and network processing.

1. INTRODUCTION

It is widely recognized that to support emerging continuous media (CM) applications having real-time constraints, operating systems should allow user applications fine-grained control over their CPU allocations. Moreover, in a multiuser, general purpose machine environment, protecting user allocations from each other and from other system activities is an important goal. It allows admitted CPU reservations to retain their quality of service (QoS) significance (e.g., delay, rate and progress guarantees) in spite of competing scheduling demand, thread synchronization, client/server interaction, interrupt processing, etc. Towards this goal, we present and evaluate experimentally three complementary kernel mechanisms, pertaining to thread scheduling, inter-process communication and receive-side network processing, respectively.

For thread scheduling, we adopt an approach based on service curve, a monotonically increasing function $S(\cdot)$ specifying the minimum amount of cumulative CPU time a thread should receive as a function of time, provided the thread has sufficient demand. A linear service curve corresponds to the traditional notion of service rate. In addition, a concave service curve is one that has a decreasing slope. By specifying an initially higher service rate, it allows applications to achieve smaller delays without having to raise their long term rates. This property can positively impact CM applications such as interactive audio, which requires low delay although it is not CPU intensive. In contrast, a convex service curve is one that has an increasing slope. It allows CPU intensive but delay insensitive applications to relax their CPU requirements, thereby allowing delay sensitive applications to meet their timing constraints. Our scheduling algorithm is adapted from the hierarchical fair service curve (HFSC) algorithm proposed for network scheduling in Ref. 16.

The scheduling algorithm aside, interesting issues arise from the implementation and integration of guaranteed CPU scheduling in a multithreaded OS kernel. This is important because apart from how we compute thread priorities, predictable performance is hard to achieve on a general purpose machine also because threads can contend for synchronization resources. The idea of priority inheritance to combat resulting phenomena of priority inversion is not new, and has been applied in the standard Solaris kernel for different dispatch levels. New performance issues...
appear, however, when thread priorities are dynamically adjusted and have QoS interpretations, and when service curve scheduling interacts with existing kernel dispatch levels.

Another form of priority inversion occurs due to the paradigm of client/server programming. Since servers are deployed without advance knowledge of the timing requirements of their clients, it is almost impossible to ensure compatible resource specifications between client and server. A more subtle problem can also occur that concerns the synchronous form of client/server programming. Since server code in this case does not run until requested by a client thread, and when it does run, the client thread blocks until the server finishes, there seems little motivation for the server to commit a separate CPU reservation.

These considerations motivate our use of a train abstraction, which extends thread level performance guarantees to local client/server computations. This is achieved by allowing a thread of control to visit multiple protection domains while carrying its scheduling and reservation state intact. In this way, server code can automatically run according to the resource needs of its clients, and there is no need to acquire a separate CPU reservation for the interaction. Train has been applied in real applications. In this paper, we present a case study of retrofitting an existing Solaris X window server to support train access.

For interrupt processing, it is hard to use service curves directly, since the necessary service curves would be hard to determine. In modern computer systems, a major source of extensive interrupt processing is packets received from the network. One possible solution is to redesign the network subsystem to minimize the use of interrupts, such as the user level protocol approach in Ref. 21. (Interrupt handling is used in Ref. 21 only to demultiplex packets to the network endpoints; receive-side protocol processing is done at the priority of the receive application.) However, mandating such a relatively major change may be difficult to achieve in all existing systems. Hence, we provide a module called Atempo\(^1\) for Internet flows to reserve CPU capacity for receive-side protocol processing. Packs arriving without the necessary reservation are then dropped early by the network interrupt handler, before too much CPU time is consumed.

Finally, we present system interfaces for users and legacy Solaris applications to access the underlying QoS support. This demonstrates practical deployment of our services in a real system environment. Experimental results demonstrate the delay and rate performance of our system under various conditions of lock contention, client/server programming, and network processing. The reported system has been in production use at Purdue for some time, supporting daily activities of our users.

1.1. Related work

Our service curve based CPU scheduler generalizes the widely used notion of progress rate employed by various other CPU schedulers offering QoS guarantees, such as the hierarchical SFQ scheduler in Goyal et a\(^2\) and the proportional-share scheduler in Jeffrey et a\(^\text{11}\)\(^1\) As discussed, it allows delay and rate guarantees to be flexibly decoupled. While the basic scheduling algorithm we employ is previously proposed by Stoica et a\(^\text{16}\)\(^1\) for bandwidth sharing in networks, we demonstrate its use in the different context of CPU scheduling for a general-purpose OS kernel. The OS environment presents some distinct challenges such as estimation of thread scheduling demands, effects of priority inversion, issues of kernel preemption, and a complex kernel synchronization structure. These issues are not relevant in Ref. 16, but are crucial for our work.

Borrowed-virtual-time (BVT)\(^7\) addresses the same problem of dispatch latency as our thread scheduler. It allows a thread to specify a warp time which effectively reduces the thread’s virtual time in relation to other threads in the system, causing the thread to be preferentially dispatched. It is not clear how performance isolation is achieved with BVT, since the effectiveness of a warp value depends on the warp values of other threads. In contrast, our scheduler provides effective admission control in which the latency benefits of an admitted service curve cannot be later undone by other threads. BVT does not require thread deadlines to be known, but limits the time during which a thread can run swapped. Our system uses deadlines, but does not require applications to predict these deadlines. Instead, we provide a method of exponential averaging that estimates the deadlines automatically. The effects of mispredicting the deadlines should be no worse than specifying too low a warp limit for BVT. Moreover, our scheduler does not perform schedulability tests like some other deadline-based schedulers, such as SMART.\(^\text{13}\) This eliminates the problem when the system predicts that some deadlines cannot be met when in fact they could be (such as if some threads are not fully utilizing their reservations).

\(^1\)From the music term à tempo – to the best.
The train abstraction synthesizes the goal for efficiency in lightweight RPC\textsuperscript{2} and Solaris door,\textsuperscript{10} and the goal for QoS performance in priority handoff.\textsuperscript{17} Similar to migrating threads,\textsuperscript{8} train adopts a mechanism of allowing a thread of control to traverse multiple protection domains without intervening rescheduling actions. Compared with migrating threads implemented for a microkernel, train is built for a more widely adopted Unix kernel design. More importantly, migrating thread has not been integrated with QoS-aware scheduling support in Ref. 8 and its benefits for CM and other soft real-time applications have not been demonstrated. Provision of QoS control across protection domains are also addressed in Resource Containers,\textsuperscript{1} Software Performance Units,\textsuperscript{18} and Reservation Domains.\textsuperscript{3} In those approaches, threads can bind to orthogonal resource objects at run time. During cross-domain calls, these resource objects can be passed from the client thread to the server thread. In comparison, train retains the use of threads as CPU resource principals, instead of relying on a more general abstraction. Train has efficiency advantages in that we do not mandate overheads in thread scheduling and passing of resource objects during remote calls. A limitation is that train can only control CPU scheduling, but not the scheduling of other system resources like memory and disk bandwidth. However, train can be used as an orthogonal mechanism to a resource abstraction that manages other resource types.

Atempo reservations for receive-side network processing are motivated by concerns such as receive livelocks.\textsuperscript{12} It is an application of the principle of early packet demultiplexing, exemplified in the lazy receive processing (LRP) approach in Ref. 6. (The path abstraction in Scout also applies the principle in protection against denial-of-service attacks.\textsuperscript{15} ) In LRP, if a flow has insufficient CPU reservation to process incoming packets, buffer overflows occur, although the size of the buffers is not explicitly controlled. With Atempo, we allow the maximum number of outstanding packets to be specified, but do not require it (some default can be used). We believe that controlling the number of outstanding packets (rather than the buffer size) is a sound design strategy, since interrupt processing is incurred per-packet, independent of the packet size. In addition, Atempo has two novel aspects. First, it allows reservations to be created as separate objects, independent of socket endpoints that access the network. Second, implemented in the buffer management subsystem used by network protocols, it requires no change to the protocol implementations themselves.

Linux/RK\textsuperscript{14} includes a component for CPU scheduling based on a reserve abstraction. Reserve capacity is depleted on CPU use, and is periodically replenished by a separate high priority system thread. The approach only relies on a static priority scheduler from the host OS, and thus achieves wide portability. In contrast, our CPU scheduler is based on dynamic priorities computed to provide performance isolation. Although we require more basic kernel changes, our approach might be more efficient since a separate system thread does not have to be used. Other approaches to portable real-time scheduling have appealed to mechanisms at the user level only, such as Chu and Nahrstedt.\textsuperscript{4} An important benefit is that the system can be readily deployed on existing computing platforms. However, it is not immediately clear how more subtle issues such as priority inheritance can be addressed with purely user-level mechanisms.

1.2. Paper organization

The balance of this paper is organized as follows. In section 2, we give an overview of thread scheduling using hierarchical fair service curves, and present its command interface and application programming interface. The train abstraction for extending thread level performance guarantees to cross domain computations is overviewed in section 3. For interrupt driven network receive, CPU time for protocol processing cannot be controlled directly by HFSC. Section 4 presents a complementary mechanism that allows Internet flows to reserve CPU resource for network receive. The Atempo interface supporting third party reservations is also introduced. In section 5, we discuss the issue of priority inheritance that arises from integration of our CPU scheduler in Solaris 2.3.1. Section 6 presents experimental results on system delay, rate and efficiency performance under various conditions of lock contention, client/server programming, and network processing.

2. DECOUPLED DELAY AND RATE GUARANTEE CPU SCHEDULING

Our CPU scheduler allows Solaris applications to reserve guaranteed CPU time. It allows CPU capacity to be recursively partitioned into configured service classes with given resource specifications. Threads can then be admitted to any configured service class with sufficient capacity, and we view them as leaf service classes (which of course cannot admit other threads). A resource specification for a (configured or leaf) service class is in the form of a service curve $S(\cdot)$, whose use is explained in section 2.1.

Any CPU service class created in our system is given a global integer identifier. A class with id $p$ can then be configured explicitly as a child of another class with id $q$ using the privileged call
join_node(p, q);

This allows to establish a CPU sharing hierarchy. Threads are similarly created to be children of specified configured service classes. A created thread can later change its resource specification, including leaving its original service class and joining a new one.

2.1. Algorithm overview

We give a high level overview of the hierarchical fair service curve (HFSC) algorithm for thread scheduling in our system. Further details can be found in Ref. 16 and Ref. 20.

HFSC uses service curve as the resource specification. A thread, say $i$, is said to be guaranteed its service curve $S_i(\cdot)$ if for any time $t'$, there exists a time $t < t'$ when $i$ becomes runnable and for which the following holds:

$$ w_i(t, t') \geq S_i(t' - t) \quad (1) $$

where $w_i(t, t')$ is the amount of CPU time received by $i$ during the interval $(t, t']$. Notice that the above condition depends not only on the service curve, but also the points in time at which $i$ becomes runnable. To handle this dynamic nature of a thread’s service requirements, we define a deadline curve $D_i(\cdot)$ which is initialized to $S_i(\cdot)$, and is updated each time $i$ becomes runnable, as follows:

$$ D_i(t) := \min(D_i(t), S_i(t - t') + c_i(t')), \forall t > D_i^{-1}(c_i(t')) $$

where $c_i(t')$ denotes the amount of CPU time $i$ has received up to time $t'$.

For each runnable thread $i$, we also maintain an estimate of its immediate CPU demand $\hat{c}_i$, which is how long $i$, if scheduled, will run until the next rescheduling point occurs. The estimate is updated whenever a block event occurs for $i$, as follows:

$$ \hat{c}_i := \alpha \times \hat{c}_i + (1 - \alpha) \times \delta_i $$

where $\delta_i$ is the elapsed time since $i$ was last scheduled after being blocked or preempted. The exponential averaging parameter $\alpha$ controls the responsiveness of the algorithm to new data samples: the smaller is $\alpha$, the more responsive is the algorithm. We currently use $\alpha = 0.25$. Using $D_i(\cdot)$ and $\hat{c}_i$, a deadline $d_i$ can be computed for runnable thread $i$, as follows

$$ d_i = D_i^{-1}(w_i(t) + \hat{c}_i) $$

where $w_i(t)$ is the total amount of CPU time received by $i$ up to time $t$. It can then be shown that if threads are scheduled in increasing order of their deadlines, then their service curves will be met, provided that CPU time is not overbooked. This gives the service curve earliest deadline (SCED) policy.

Notice that because threads can become runnable at different times, it is in general impossible to serve all threads at the rates of their service curves at all times. (This is not a weakness of HFSC, but a fundamental limit of any delay scheduler in which task release times are not controlled.) To see why, denote by $<m_1, d, m_2 >$ a two piecewise linear service curve having slope $m_1$ from $[0, d]$ and slope $m_2$ from $(d, \infty)$ ($d$ in ms). Consider two threads $P$ and $Q$ with service curves $<0.1, 10, 0.9 >$ and $<0.9, 10, 0.1 >$, respectively. $P$ becomes runnable at time 0, and $Q$ becomes runnable at time 10 ms. From 10-20 ms, therefore, the aggregate service curve of $P$ and $Q$ has rate 1.8, which exceeds the CPU capacity. Clearly, $P$ and $Q$ cannot both run at rate 0.9.

To satisfy service curves, therefore, it is generally needed to provide service in advance for some threads, so that irrespective of future CPU demands, thread deadlines are not in danger of being violated. Providing too much advance service, however, may unnecessarily jeopardize fairness. To solve this dilemma, a key observation in Ref. 16 is that it is possible to slightly overestimate (for efficient implementation) the minimum amount of advance service each thread should receive so that deadlines are not in danger of being missed. A thread that has not received its minimum advance service is said to be eligible. When all threads in the system are ineligible, therefore, we can schedule to optimize for fairness. Otherwise, we prefer the real-time goal by first scheduling eligible threads in SCED order.

Complementary to the real-time goal, fairness aims to minimize normalized service discrepancies between sibling nodes in the CPU sharing hierarchy. This ensures that runnable nodes make progress in ratio of their service curves. To do so, the system keeps a virtual time $t^*$ that remembers the amount of CPU time a service class or thread has received normalized by its resource specification. To schedule threads according to the fairness criterion, we recursively select a child node having a minimum virtual time. Service time received by the thread increases the virtual times of all its ancestor nodes. Together with update of deadline and eligibility curves at rescheduling points, this gives a form a rate control in which thread priorities are adjusted according to CPU usage.
2.2. CPU scheduling interface

Solaris supports different scheduling classes. Our system retains the use of the SYS class for interrupt processing (see also section 5). Our service curve based CPU scheduling is implemented as a new scheduling class called HFSC. At system startup time, we partition CPU capacity into two default service classes: HFSC.C0 and HFSC.C1, that have 90% and 10% of the CPU respectively. Once the system is booted up, a privileged user can repartition the CPU into other configurations.

We change the Solaris kernel to run the init process (process number 1 and an ancestor of all Unix user processes) in HFSC.C0 with the linear service curve of slope 0.001. In Unix SVR4, children processes forked by a parent process inherit the scheduling class and parameters of the parent by default; therefore, all user processes (and their associated threads) also run in HFSC.C0 with the linear service curve of slope 0.001 by default.

We provide two principal ways to change the default scheduling parameters of an HFSC thread. First, we support the `priocntl(1)` command for HFSC scheduling. Using this command, any standard Solaris application can be started with specified parameters from a Unix shell, as follows:

```
priocntl -e -c HFSC -r1 <m1> -x <d> -r2 <m2> -n <class> <program>
```

where `<program>` is the name of the application to run, `<class>` identifies the service class (e.g. HFSC.C0) that is to be the parent of the application, and the service curve for `<program>` is specified to have slope `<m1>` (in unit of 0.1%) during time `[0, d)` (`<d>` is in μs) and slope `<m2>` during time `[d, ∞)` (i.e. we restrict the specification to only two piecewise linear service curves).

Second, application threads once started can change their scheduling parameters using the SVR4 priocntl(2) system call. The PC_SETPARMS entry point of priocntl sets the scheduling parameters of a specified set of threads (e.g. all threads belonging to a given process, the thread for the current LWP, all threads belonging to a given user, etc).

3. CROSS DOMAIN SCHEDULING

OS services are frequently implemented in isolated protection domains. This has many advantages, including modularity, protection, and service access control. A consequence, however, is that an explicit interprocess communication (IPC) mechanism will be needed for client processes to invoke services in a server domain. In this regard, the remote procedure call (RPC) is a particularly attractive mechanism which allows high level remote code access in the style of local procedure invocations.

Traditional RPC uses independently scheduled server threads to process client requests. These server threads are generally oblivious to the progress requirements of their clients. As such, they may cause forms of priority inversion, and client timing constraints may be violated. For example, a high priority client thread making a call to a low priority server thread can be indirectly blocked by another medium priority thread. In recognition of these problems, we use `train` as a new IPC mechanism for QoS provisioning across protection domains. `Train` allows a thread of control to flexibly change its process context at runtime. Hence, the thread can access remote code and process resources (e.g., opened files) without unnecessarily changing its resource and scheduling states.

The `train` API has six major functions (further details are given in Ref. 20). Among them, `train_create()` allows a server to create a train object in the file system name space that can be opened by client processes. The train object specifies a secure entry point to server code (i.e. a program counter value). A previously exported train object can be later revoked with the `train_delete()` call.

Given proper permissions, a client process can obtain a handle to a train object using `train_open()`. The handle can be passed to `train_call()` together with other user parameters. During `train_call()`, the caller thread first locates an available server stack for the call and copies in parameters from user to kernel space. It then switches to the resource context of the server process that exports the opened train object, and sets up execution context on the server stack so that it will begin execution with the exported entry point. When the server function completes, it calls `train_return()`, which allows the caller thread to return to the process context at the time the corresponding `train_call()` was made. When a process finishes using a train object, it gives up its reference to the object through `train_close()`.

Straightforward source code changes can enable legacy applications to switch from a traditional IPC mechanism to `train`. We have applied it to a Solaris X window 11 server. Section 6.4 describes the modified system and evaluates its performance.
4. RECEIVE NETWORK RESERVATION

HFSC scheduling does not explicitly control the CPU demand of interrupt activities. Particularly important is interrupt processing due to network packet arrivals.\textsuperscript{6,12} If such processing overhead cannot be controlled, then aggressive network flows generated by greedy or malicious remote applications can gain a grossly unfair share of CPU time.

We provide an Atempo module for users to explicitly reserve system resources used in receive side-packet processing. A reservation is of the form $\langle \text{flowspec} \rangle \langle \text{rspec} \rangle$, where $\langle \text{flowspec} \rangle$ corresponds to an Internet flow specification, and $\langle \text{rspec} \rangle$ specifies the reservation amount, which is the maximum number of outstanding packets that can be queued for flows classified to the flow specification. (A packet is \textit{outstanding} if it is being buffered inside the kernel waiting to be read by an application.) Hence, if packets are destined for a process which does not have enough CPU reservation to consume its packets, these packets can be dropped early by the system, limiting CPU processing overhead. Similarly, stray packets with no intended receivers are also discarded early, if they classify to a null Atempo reservation. This is an application of the design principle of early packet demultiplexing.

Atempo works as follows. When a network receive interrupt occurs for an IP packet, the interrupt handler examines the received packet’s IP header and classifies it to a \textit{most specific} Atempo reservation. (The classification mechanism is being implemented as a hash table with one-behind caching, which scales well to a fairly large number of reservations.) If there is non-zero capacity in the reservation to admit the packet, the packet consumes one unit of the reservation and is passed up to the protocol stack. Otherwise, the packet is dropped immediately. If the packet is passed up, we store a reference to its Atempo reservation with the network data buffer holding the packet. This achieves efficiency, since the system needs future access to the reservation. The unit of reservation consumed by the packet is not replenished until the buffer is later freed by the system. If the free occurs on the normal read path of the stream head, an application has read the packet in question. The packet’s reservation unit is then immediately returned to the corresponding Atempo reservation. If the free occurs because there is no receive endpoint for the packet, then the reservation unit is not returned. Atempo reservations can be created, deleted, changed or listed with the \texttt{atempo(1)} command.

In our implementation, network data buffers can reference Atempo reservations. When a data buffer is duplicated (using the Solaris \texttt{dupb()} call) that has such a reference, the reservation unit in question is not returned until the last reference to the data buffer is freed. If a data buffer is copied (using \texttt{copyb()}), its Atempo reservation, if any, is \textit{not} copied. Atempo reservations that are being referenced by any data buffers cannot be deleted until the last such reference is gone. The system automatically enforces this condition.

4.1. Network Reservation Interface

Atempo supports third-party resource reservations for receive side network processing. Hence, reservations to be used by an application need not be made by the application itself, but possibly by software agents specialized for resource management. The \texttt{atempo(1)} command allows privileged users to create such reservations from a Unix shell:

\begin{verbatim}
   atempo \langle \text{flowspec} \rangle \langle \text{rspec} \rangle
\end{verbatim}

where $\langle \text{flowspec} \rangle$ specifies an Internet flow specification and $\langle \text{rspec} \rangle$ specifies the desired reservation amount. The \texttt{atempo} command invoked with no parameters lists all the current Atempo reservations in a system.

5. INTERRUPT PROCESSING AND DYNAMIC PRIORITY INHERITANCE

While all CPU processing should ideally take place in the context of HFSC threads, it seems impractical to have exclusively HFSC scheduling in a system. It is difficult, for example, to process interrupts using HFSC, since the needed service curves are hard to determine. Moreover, interrupts are typically designed to obey a certain priority order which is hard to control precisely with HFSC. If interrupt processing is unnecessarily delayed, incorrect system behavior may result.

In view of the above, we retain the Solaris SYS class for interrupt threads.\textsuperscript{5} Moreover, we let interrupt threads have strictly higher priorities than any other threads in the system, and keep the relative priorities between interrupt threads the same as the relative interrupt levels handled by the threads. While all runnable HFSC threads are

\textsuperscript{5}In Solaris, interrupt processing occurs in interrupt context although interrupt threads do not need to run as full fledged threads – thus improving efficiency – unless they block.
maintained within an HFSC specific priority queue, each of the interrupt priorities keeps its own dispatch queue of threads as in the original Solaris kernel.

This gives rise to another concern. It has been observed that real-time performance is hard to achieve not just because thread priorities have to be computed in an appropriate manner, but also because threads can contend for resources, which upsets their intended priorities. Priority inheritance is essential to cope with this problem, whereby a lower priority thread blocking a higher priority one should inherit the latter’s priority. This complicates our HFSC scheduler because an HFSC thread blocking an interrupt thread will inherit an interrupt level priority. The inheriting thread should then “leave” the HFSC priority queue and “join” an interrupt level dispatch queue. There are two basic approaches in which such leave/join can be performed.

The first approach is the following. We temporarily remove the inheriting thread from HFSC while the priority inheritance is in effect, and let the thread rejoin HFSC once the original priority is restored. However, doing so means that the thread will be immune to all forms of HFSC rate control (which would be performed at given rescheduling points) while it has the inherited priority. This essentially allows the thread to use the CPU “for free” during certain time intervals. Section 6.3 presents some experimental results that demonstrate the resulting impact on real applications.

In the second approach, we remove an HFSC thread from the HFSC priority queue when the thread is inheriting an interrupt level priority. (The removed thread will then join the appropriate SYS level dispatch queue.) However, we continue to perform usual rate control for the inheriting thread at all rescheduling points. We say that the inheriting thread is physically inactive in HFSC (i.e. it is temporarily not scheduled according to HFSC criteria) but is logically active (in that it will still affect the scheduling state of certain HFSC data structures). For example, all logically active HFSC threads together define a logically active fair sharing hierarchy. An internal sharing node in this hierarchy is logically active if any of its children is logically active, and its virtual time (for fair sharing) can be computed according to normal HFSC rules. An experimental evaluation of this second approach versus the first approach is given in section 6.3. Implementation of a proper priority inheritance strategy is an important factor contributing to the stability of our system.

Further issues about the CPU scheduler implementation, including locking of dispatch data structures and reservation release strategy, can be found in the technical report.\textsuperscript{19}

6. PERFORMANCE EVALUATION

Our current system reported in this paper has been in production use by members of the System Software and Architecture Lab at Purdue for several months. It is being run on a cluster of Ultra-1 and Pentium II machines interconnected by Ethernet, FastEthernet and Myrinet interfaces. Common tasks performed by our users include Web browsing, Real audio and video streaming, program editing and compilation, document processing, email, network access with telnet and rlogin, playing of MPEG3 songs, etc. When users log on to our system without explicitly using its underlying support for QoS, they generally are unaware that a modified Solaris kernel is being used, and see the same level of performance as with a standard kernel. This gives us reassurances that QoS support does not need to be intrusive, such as reducing system flexibility, limiting user tasks, or compromising ease of use. If desired, however, users can easily find out the scheduling dispositions of their processes with the priocntl(1) command. Figure 1 shows the output of an example call. As shown, all user processes run by default in the HFSC class with a linear service curve of rate 0.1% and a preemption quantum of 1 ms.
We now report experimental results to illustrate the performance of our system when its QoS features are invoked. Measurement data were taken on a Sun Ultra-1/2bus workstation with a 167 Mhz processor, 512 Kbytes of E-cache, and 128 Mbytes of main memory.

6.1. Service curve sharing dynamics

Our first experiment illustrates some sharing dynamics with service curves. We use a CPU intensive application numeric that repeatedly does rounds of some mathematical computation and prints a timestamp after each round. In the first experiment, we ran three (single threaded) processes of numeric concurrently: the first process with a convex service curve <1%,3 seconds, 9%>, the second process with a linear service curve of 4.5%, and the third process with a concave service curve of <9%, 3 seconds, 1%>. Figure 2 plots the round number against its corresponding timestamp for each process. Since the three processes became runnable at about the same time and remained runnable afterwards, they simply made progress in ratios of their service curves. Notice that although the concave service curve had a much lower long term rate than the convex service curve, its shape allowed its process to initially (following a wakeup event) receive much more CPU time. This is a key factor that leads to flexible decoupling of delay and rate performance. (Because the total CPU rate of all threads in this case is 14.5%, the third process runs at its higher rate for about 14.5% x 3 seconds, or 435 ms.)

6.2. Decoupled delay and rate performance

To further assess the delay performance of our system, we use two UDP applications, one for send and one for receive. The send application sends a packet to UDP port 10000 on a specified host at three second intervals. Each packet contains 10 bytes of user data. The receive application reads from UDP port 10000. For each packet received, it performs some computation instrumented to take about 30 ms on our Ultra-1 measurement platform, and sends back a reply UDP packet with 10 bytes of user data. (Hence, the receive application requires a low long term CPU rate.) In an experiment, we ran the send application on an Ultra-5 and the receive application on an Ultra-1, with the two machines connected to a same 10 Mbps Ethernet subnet. The send application addressed packets to the Ultra-1; these packets were thus read by the receive application. Because the Ultra-5 was very lightly loaded, the send application was able to provide an independent and timely stream of packet arrivals on the Ultra-1.

To accurately quantify the delay from the time a packet arrived on the Ultra-1 to the time that the receive application was able to process it and send back a reply, we inserted some simple but effective measurement code in the Ultra-1 kernel. Specifically, early at the receive network driver, we inspected the header of an arriving Ethernet packet. If it was determined to be destined for UDP port 10000, we recorded a timestamp of the arrival using the gethrtimer() call, a high resolution timer with about 4 µs precision. At the send network driver, we similarly determined if the Ethernet packet to send came from UDP port 10000. If so, we used gethrtimer() to record a timestamp for the send. The difference between a receive timestamp and its corresponding send timestamp gives a measured delay value, which accurately accounts for any scheduling delay the receive application experienced before it could respond to an external packet arrival event. One hundred delay samples were generated over a 5 minute period as packets were sent at 3 second intervals.
Table 1. Delay statistics for two experimental runs with linear and concave service curves, respectively. All numbers are in ms.

<table>
<thead>
<tr>
<th>Run</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>S.D.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear 2%</td>
<td>30.91</td>
<td>1931.89</td>
<td>1327.96</td>
<td>755.05</td>
</tr>
<tr>
<td>Concave 90%/2%</td>
<td>31.27</td>
<td>41.80</td>
<td>39.35</td>
<td>4.18</td>
</tr>
</tbody>
</table>

Figure 3. Priority inheritance approach in which an HFSC thread having an inherited SYS level priority temporarily leaves the HFSC class. This results in rate fluctuations over small time scales.

To show how service curves can allow the receive application to achieve different delays in the presence of competing CPU intensive applications, we did two separate runs of the experiment. In the first run, the receive application ran with a linear service curve of 2% CPU capacity. Simultaneously, a CPU intensive competing workload ran with a linear service curve of 98%. The actual CPU load was 100% throughout the experiment (as shown by the Solaris `perfmon` application). The first row in Table 1 shows the maximum, minimum, average and standard deviation for the 100 delay samples collected in this run. We observe that because the receive application ran with a low linear CPU rate, it also experienced high scheduling delay on the order of one to two seconds.

In the second run, we ran the receive application with a concave service curve, with rate 90% during the first 50 ms, and rate 2% after 50 ms. The competing CPU workload ran with a convex service curve of 10% during the first 50 ms, and rate 98% after 50 ms. As in the first experiment, the actual CPU load was 100% throughout. As shown in the second row of Table 1, the higher initial CPU rate specified by the concave service curve was effective in reducing the scheduling delay for the receive application. In this case, the average delay was lowered significantly to 39.35 ms (the maximum and standard deviation being 41.80 and 4.18 ms respectively).

6.3. Priority inheritance

This section evaluates the two approaches of priority inheritance discussed in section 5. In our experiment, we ran three single-threaded `mpeg2play` processes decoding (without display) a same segment of MPEG IPPPP video together. One process had rate 1% and the other two had rate 8%. We show progress for the three processes by plotting the time at which a frame is displayed against the frame number.

Figure 3 shows the results for the first approach in which an HFSC thread inheriting an interrupt level priority is temporarily removed from the HFSC class. The figure shows that while the processes made long-term, coarse time scale progress in their relative rate ratios, their short term progress rates exhibit considerable irregularities.

Figure 4 shows progress for the three processes under the previous experimental setup but using the second approach in which an HFSC thread inheriting a SYS level priority remains logically active in HFSC. As shown, the progress rates were able to maintain their intended ratios over much shorter time scales than in the previous experiment. We conclude that continuing to perform usual rate control for HFSC threads that have inherited interrupt level priority achieves stable performance.

*Notice, however, that the long term CPU rate remained the same at 2%.
Figure 4. Priority inheritance approach in which an HFSC thread having an inherited SYS level priority is physically inactive in HFSC, but remains logically active. The threads are able to make progress in their intended rate ratios over finer time scales than the previous approach.

6.4. Video display with train

We have extended Xsun, Solaris server for X Window version 11, and libXext.so, the X client extension library, to include train support. Two new extension library functions are supported. The first one is

```
TrainHandle xTrainShmAttach(Display *dpy, XShmSegmentInfo *shminfo);
```

which works similarly as the standard extension library call XShmAttach in that it establishes a shared memory display between client and server. In addition, however, xTrainShmAttach returns a handle that identifies the calling client’s connection in the X server. The client can then later use this handle to make train requests with the server.

The second added function is

```
Status XTrainPutImage(TrainHandle handle, Display *dpy, Drawable d, GC gc,
                     XImage *image, int src_x, int src_y, int dst_x, int dst_y, u_int src_width,
                     u_int src_height, Bool send_event);
```

XTrainPutImage works similarly as the standard extension library call XShmImage in that it displays an X image in a (previously established) shared memory display. There are, however, two differences. First, XTrainPutImage additionally takes a TrainHandle, previously returned by XTrainShmAttach, as first parameter. Second, XTrainPutImage uses train to directly access the server function, whereas XShmPutImage uses a Unix domain socket to send requests to the separately scheduled X server.\footnote{Requests passed by domain sockets are small in size; image pixels are passed by shared memory.}

To support train access, the X server is made multithreaded. To synchronize between client threads, a mutex lock is acquired at service entry and released when the service completes. At startup time, the X server exports the XTrainPutImage entry point as /tmp/.Xtrains. Using the /tmp directory is natural since there is typically one /tmp directory and one X server on each machine.

Experimental evaluation

We have experimentally evaluated the performance of X window display using train. In our experiment, we ran three processes of the `mpeg2play` application concurrently, which all play a same MPEG IPPPP encoded video segment. The first and second processes used standard Unix domain sockets (specifically XShmPutImage) to call to the X server, for display of decoded pictures in an X window. The first process had rate (i.e. linear service curve) 1% and the second process had rate 8%. The third process also had rate 8%, but used XTrainPutImage to display pictures in an
Figure 5. Progress rates of three mpeg2play threads decoding MPEG video and displaying frames in an X window.

Figure 6. Distribution of times between frames displayed by X for a client thread, when Unix domain socket is used as the IPC mechanism.

X window. Because of the X server’s synchronization strategy, the three mpeg2play processes compete for a same mutex lock. The X server ran with rate 1%.

Figure 5 shows that in spite of the lock contention, the three processes made progress at roughly their relative reserved rates. This is because the Solaris kernel enqueues threads waiting for a synchronization resource in decreasing order of their thread priority, and when a resource becomes available, the thread at the head of the wait queue gains access. Hence, a client thread’s reserved rate is also a good estimate of the rate at which the thread can successfully obtain service from the X server.

Nevertheless, we can see a clear benefit of using train for predictable IPC in this experiment. To substantiate the point, Figure 6 presents a distribution of the times between frames displayed by the X server for the client thread using Unix domain socket. The minimum, maximum, mean and standard deviation of the inter-frame times were 15.1, 136.1, 44.8 and 23.0 ms, respectively. Figure 7 gives the corresponding distribution for the client thread using train. The minimum, maximum, mean and standard deviation of the inter-frame times in this case were 17.8, 98.8, 43.7 and 13.1 ms, respectively. For Unix domain socket, because the X server ran with a much lower rate than its client, a significant number of frames were delayed by about 50–100 ms more than the rest. For train, the higher delays were not observed, because the server was able to run with the same reservation provided by its client. In terms of perceived video quality, train did result in visually smoother playback.

We report the above experiment to illustrate the subtle effects of lock contention. The performance contrast between train and Unix domain socket is much more stark if the application that runs concurrently with mpeg2play
does not compete for the X server lock. To illustrate, we ran mpeg2play with X display (as fast as possible) concurrently with a "greedy" application that never blocks. With train, we used a CPU rate of 8% for both mpeg2play and greedy. We observed smooth video quality comparable to a high-rate mpeg2play in the previous experiment. With Unix domain socket, using 8% rate for mpeg2play and greedy but 1% rate for the X server, the visual quality of the video became very choppy. This was because the X server was too slow, causing its request queue to overflow, and a significant fraction of the pictures failed to display.

In terms of efficiency, we have shown\textsuperscript{20} that the train implementation of a simple adder service is about ten times faster than the same service implemented using Solaris RPC. Please also refer to the technical report\textsuperscript{19} for statistics of a detailed breakdown of train call and return path overheads.

6.5. Network receive reservation

For this set of experiments, we configured two Atempo reservations: the first reservation for UDP destination port 8000 has value $\infty$ (i.e. no limit on receive-side CPU processing for packets destined for this UDP port), while the second reservation for UDP destination port 8001 has value 3. In an experiment, we sent a continuous UDP stream of packets at a high rate of 10 Mbps from a Pentium II to an Ultra-1. The two machines are connected to a same isolated Ethernet subnet. In each experiment, the UDP stream consists of packets all having the same size of user payload. We varied this size to be 1400, 1000, 500, 100 and 1 bytes in five separate runs destined for UDP port 8000. These five runs are then repeated for UDP port 8001. In all the experiments, two applications reading from UDP ports 8000 and 8001 respectively were blocked throughout and hence were not able to read any of the received packets.

Using perfmon, we sampled the resulting CPU load on the Ultra-1 at one second intervals in an experiment. (The 10 Mbps UDP stream was active throughout the time these samples were taken.) The samples averaged over a 60 second period of measurement are plotted in Figure 8 (with corresponding standard deviation indicated as a range around the average value). The figure clearly indicates much reduced CPU load on the Ultra-1 for UDP port 8001 than for port 8000. This is because the limited reservation for port 8001 causes packets to be discarded early before they consumed too much CPU time. Atempo protection was quite good when the UDP payload was 500 bytes or larger (about 2% to 4% of CPU time was consumed within this range). At 100 bytes, about 12% of CPU time was consumed, and at one byte, about 25% of CPU time was consumed. While these numbers are significant, they nevertheless represent much improved performance over the case of no reservation limit, in which as much as 95% of CPU time can be consumed for a one byte packet size.

7. CONCLUSIONS

We presented three complementary mechanisms designed to give user applications predictable and fine-grained CPU access, related to thread scheduling, inter-process communication, and receive-side network reservations, respectively. We discussed algorithm design and integration into a truly multithreaded OS kernel, including support issues for legacy applications. Experimental results from a prototype implementation showed the effectiveness of our techniques in various situations. We conclude that our system achieves highly effective rate and delay performance for
Figure 8. Performance of Atemo reservations. Packets destined for UDP port 8001 are processed with a limited Atemo reservation of value 3. Packets destined for UDP port 8000 are processed with an unlimited reservation.

real multimedia applications, in the face of priority inversion problems arising from lock contention, client/server interaction, and aggressive incoming network flows.

REFERENCES

10. Sun Microsystems Inc., Solaris 2.5.1 online clean(2) system call documentation.