Processor Capacity Reserves:
An Abstraction for Managing Processor Usage

Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
{cwm,savage,hxt}@cs.cmu.edu

1. Motivation

Multimedia applications require operating systems that support time-constrained data types such as digital audio and video. These *continuous media* [1] demand timely service from the system, and time-sharing scheduling algorithms are not sufficient. Furthermore, simple fixed priority scheduling, used in many hard real-time systems, does not necessarily guarantee the successful execution of arbitrary collections of programs which may have conflicting timing requirements or which may overload the system [7].

We have designed and implemented a processor capacity reservation mechanism to allow programs to reserve the capacity they need to run. Enforcement of reservations provides a scheduling firewall to protect programs from competition for the processor; this is similar to the firewall provided by memory protection, which isolates programs from outside interference in their address spaces. Using reservation, a system can control processor capacity allocation in the same way that it controls discrete resources like memory or storage space, and this prevents over-committing the processor.

Our reservation mechanism depends on a scheduling framework where each reservation is expressed as a rate of progress (defined as computation time per period of real time). A rate can be associated with non-periodic programs as well as periodic programs. An admission control policy, based on rate monotonic scheduling theory [5, 6], moderates access to processor resources. An enforcement mechanism ensures that the reserved computation is limited to the requested rate and does not interfere with other activities. The reservation mechanism measures the processor usage of each reserved program and even includes in that measurement any computation time consumed by servers invoked on that program's behalf. A detailed description of the scheduling framework, the admission control policy, and the enforcement mechanism appears elsewhere [7].

Supporting multimedia applications is our primary goal in this work. Many multimedia systems assume that this kind of processor reservation functionality is available [1, 2], but previous reservation systems typically use extended time-sharing algorithms which attempt to match long-term utilization with preset reservation values [3, 4]. These systems do not have the accurate usage measurement and control required by low-latency multimedia applications.

A reservation mechanism like ours is also useful in other application areas. Multiprocessor systems can use reservations to interleave parallel programs on a collection of processors. The accurate processor

---

This work was supported in part by a National Science Foundation Graduate Fellowship, by Bellcore, and by the U.S. Naval Ocean Systems Center under contract number N00014-91-J-4061. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NSF, Bellcore, NOSC, or the U.S. Government.

usage statistics gathered by the reservation mechanism can be used for performance tuning since they provide information about the processor usage of user-level servers invoked by the threads being analyzed. A reservation mechanism may also prove useful in dynamically configured, responsive hard real-time systems that utilize off-the-shelf programs which must be protected from timing interference by other programs.

We have implemented an initial prototype of the processor capacity reservation system in Real-Time Mach [8] which is a compile-time extension to Mach 3.0. Real-Time Mach provides extended scheduling and timing support which made scheduler modifications convenient, but these extensions were not essential to our implementation. We expect that our design could be implemented in other microkernel systems with comparable effort.

Our work demonstrates the feasibility of implementing an effective reservation system, but it is not yet general enough to handle arbitrary interactions among programs and devices. So far, we have concentrated on reservations for arithmetic computations, standard I/O, and RPC-style IPC. Other issues, such as the effects of paging, DMA, caching, and interrupt processing, are not addressed in our current prototype, although we intend to extend the system to account for these issues.

In this paper, we briefly describe the prototype, and we demonstrate how the reservation system can be used to control processor usage in several different situations. Our work shows that we can successfully enforce reservations in the operating system and protect programs from timing interference from other programs, and that this functionality is compatible with a typical microkernel operating system architecture.

2. Implementation overview

Our implementation adds a new *reserve* abstraction to the Real-Time Mach microkernel along with reservation enforcement, and we added a new scheduling policy that supports reservation.

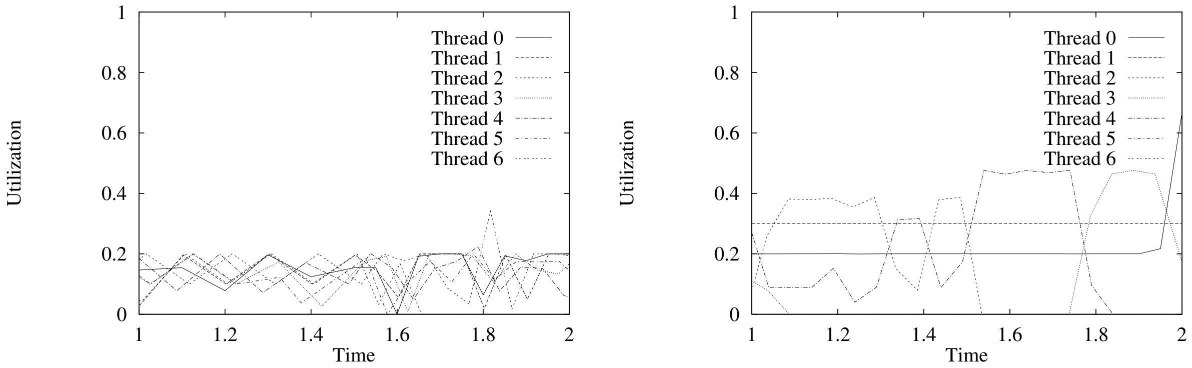
Reserves

A reserve is a kernel managed entity which represents access to processor capacity. A reserve contains a computation time and a reservation period that specify a rate of progress. This information is bound to the reserve through a reservation request which the system may accept or deny. If the reservation is denied then the program may either request a lower rate or choose to execute unreserved under a time-sharing policy.

A thread bound to a reserve is entitled to consume, in total, the reserved amount of computation time in each interval specified by the reservation period. The scheduler provides preferential service to threads with reservations and schedules such threads in order of increasing reservation periods; shorter periods have higher priority. This policy is based on the rate monotonic scheduling algorithm which is used in many hard real-time systems [6]. When a thread executes, the time it uses is subtracted from the allocation held in its reserve. When this allocation reaches zero, the thread no longer receives preferential treatment, and if it requires additional computation time, it is scheduled under the time-sharing policy. At the end of each reservation period, the reserve gets a new allocation of computation time which can be consumed during the subsequent reservation period.

Reservation enforcement in microkernels

Microkernel systems pose a problem for processor reservation since many operating system services are implemented as user-level servers. These servers are scheduled independently of their clients, and their resource usage is usually measured independently of their clients. Consequently, simply binding a



(a) Time-sharing

(b) Reservation

Figure 1: Time-Sharing Scheduling vs. Reserved Scheduling

client thread to a reserve and charging the reserve for that thread's processor usage is not sufficient. The processor usage of servers that execute on behalf of a client must be charged to the client's reserve. Our approach is to pass a client's reserve to the server invoked during an RPC. The server then charges its computation time to the client's reserve.

3. Performance evaluation

Our prototype implementation of processor capacity reserves in Real-Time Mach demonstrates the feasibility of our design. We can effectively control the computation rates of arithmetic computations, computations that use standard I/O, and computations that use simple RPC. As the work progresses, we expect to be able to measure and control all types of computation and interaction in the system.

The following test cases illustrate the behavior of the reservation mechanism in several different kinds of situations. We ran these programs on a Gateway2000 33MHz 486-based machine with 16MB of RAM and an Alpha Logic STAT! timer board which was used for precise time measurement and reservation enforcement. Our tests were run on a modified version of Real-Time Mach version MK78 using the CMU UNIX server version UX39.

Each thread in each test case consisted of an infinite loop which did arithmetic calculations. In the tests, we varied the number of threads, the binding of threads to reserves, and the scheduling policy. We recorded the computation time usage of each thread at the end of the reservation period of its reserve (50 ms in all cases), and we graphed the utilization of each thread over a one-second test duration. Other programs such as the utilization monitor, the UNIX server, and various system programs were running on the system, but for clarity, we did not include their utilization in the graphs. We minimized the effect of these programs by minimizing the interaction between the test programs and the rest of the system for the duration of the measured test, but we did not eliminate outside interference entirely.

Figure 1(a) shows that the time-sharing policy makes it difficult to predict the short-term utilization of any particular thread. The figure plots the utilization of seven threads under the Mach time-sharing scheduling policy. Each thread's usage oscillates as it goes through various phases of the multi-level feedback queueing policy, and the behavior may change radically over time as the policy collects more information about the particular set of programs that is running and tries to adjust the priorities accordingly.

In Figure 1(b), we show that our reservation system can ensure a minimum rate of computation for reserved programs. Two of the threads in the figure have reservations: Thread 0 reserved 20% of

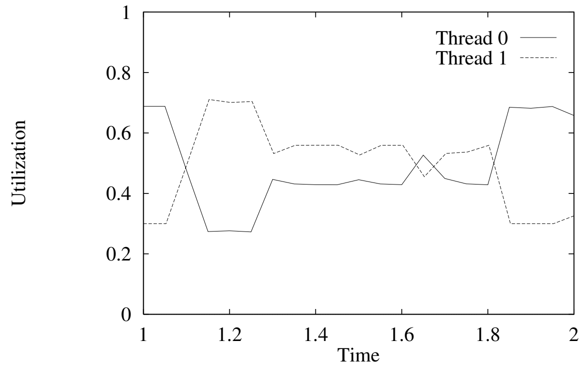


Figure 2: Reserved Scheduling with Additional Usage

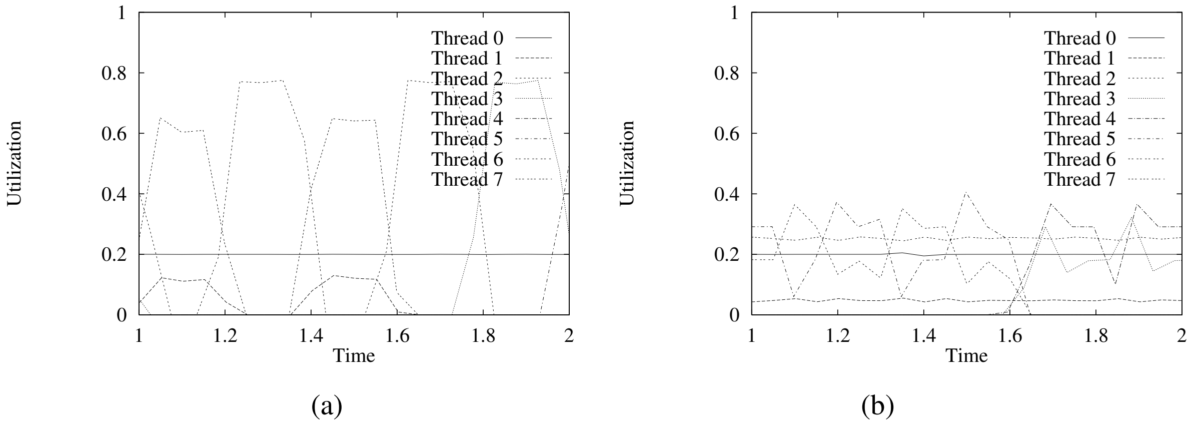


Figure 3: Uncoordinated Client/Server Reserves vs. Integrated Client/Server Reserves

the processor, and Thread 1 reserved 30%. The horizontal line at 20% and the line at 30% give the utilization of these two threads. We note that since programs are scheduled in a time-sharing pool after their reservations are consumed, a reservation ensures a lower bound on the usage for a thread which never blocks itself. This explains the increase in the usage of Thread 0 (reserved at 20%) which jumps at time 1.95 as a result of Thread 0's participation in the time-sharing scheduling pool. Five other threads are unreserved, and they each get significant processor utilization as the time-sharing policy increases their priority, and then get very little utilization as the policy decreases their priority.

Figure 2 illustrates how reserved threads can receive extra computation time by participating in the time-sharing pool after their reservations are depleted. In this figure, Thread 0 is reserved at 20%, but at time 1, its utilization is at about 70%. Thread 1 is reserved at 30%, and its utilization at time 1 is about 30%. There are no additional threads to compete under the time-sharing policy. The threads alternately get more and less processor time in addition to their reservations, but neither thread's utilization ever goes below its reserved capacity.

Figure 3 shows that charging a server's computation time against the reserve of the invoking client is essential to the correct operation of the reservation system. In the first case, the reserves of a client and server are not coordinated during an RPC, so reservations cannot be effectively enforced. The second case integrates the client/server reserve handling by charging server computation time to the invoking client.

Both of these tests have Thread 0 reserved at 20% doing an arithmetic computation, Thread 1 reserved at 30% doing a call to a user-level server (labeled Thread 7) which does an arithmetic computation, and five additional unreserved threads doing arithmetic computations.

In Figure 3(a), Thread 0's utilization is fairly constant at 20% as expected. However, the utilization of the activity associated with Thread 1 (including the server computation) exhibits some undesirable behavior. Since the reserves of Thread 1 and the server are independent, the scheduler attempts to enforce the reservation for the computation in Thread 1 alone with no regard for the server, which in this case competes in the time-sharing pool. Thus, the computation in Thread 1, which is just a call to the server, never comes near the reserved level. The server, which carries out the bulk of the computational activity associated with Thread 1, is at the mercy of the time-sharing policy. The result is that Thread 1 executes when its server can get scheduled by the time-sharing policy, and it does not execute at all when the time-sharing policy reduces the priority of the server. In the figure, Thread 1 and its server execute during the time from 1.0 to 1.25; the utilization of Thread 1 is given by the line which rises to the 12% level and then falls to 0 in this interval, and the utilization of the server is given by the line which rises to slightly more than 60% and then falls back to 0 during the interval. Neither execute during the period from 1.25 to 1.35, and then they both execute in the same pattern during the interval from 1.35 to 1.65. Clearly, the uncoordinated reservation system is not having the effect of reserving computation time for Thread 1 and its associated server.

In contrast, Figure 3(b) shows that the reservation for the activity associated with Thread 1 can be enforced by passing Thread 1's reserve to the server during an RPC. Thus, in a sense, the server computation is reserved and scheduled as an extension of Thread 1's computation. In the figure, the utilization of Thread 0 is constant at 20% as before. The usage of Thread 1 by itself is fairly constant at a utilization level of about 5%. The usage of Thread 1's server is fairly constant at a utilization of about 25%. So the combined activity receives its reservation of 30%. The time-sharing threads compete among themselves while the reserved threads receive the appropriate computation time.

4. Conclusion

We have briefly motivated and described processor capacity reserves and our prototype implementation of reserves in Real-Time Mach. The reservation mechanism provides a way for application programs to specify their reservation requests and incorporates a scheduling framework which supports an admission control policy. The reservation system accurately measures processor usage of individual threads, even when these threads invoke user-level servers to do work on their behalf, and the scheduler enforces reservations by relegating reserves to a time-sharing pool after their reservations have been consumed.

Acknowledgements

The authors would like to express their appreciation to the following people for their comments and suggestions: Brian Bershad, Ragunathan Rajkumar, Barry Hatton, and the members of the ART group and the Mach group at CMU.

References

[1] D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for Continuous Media in the DASH System. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 54–61, May 1990.

[2] D. Ferrari and D. C. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communication*, 8(3):368–379, April 1990.

[3] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1858, October 1984.

[4] J. Kay and P. Lauder. A Fair Share Scheduler. *CACM*, 31(1):44–55, January 1988.

[5] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[6] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM*, 20(1):46–61, 1973.

[7] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Technical Report CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, May 1993.

[8] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, October 1990.