Multimedia on Multiprocessors: Where's the OS When You Really Need It?

Jason Nieh[1,2] and Monica S. Lam[1]
[1]*Computer Systems Laboratory, Stanford University*
[2]*Sun Microsystems Laboratories*

Abstract

Due to the limitations of current operating systems in supporting multimedia applications, much work has been done to provide resource management mechanisms to address this problem. As processor cycles are often the most oversubscribed and critical resource, most of this work has focused on uniprocessor scheduling. However, hardware platforms are moving to multiprocessor systems, and little work has been done to address the problem of supporting multimedia applications in a multiprocessor context. This paper proposes a new multiprocessor scheduler designed to meet the requirements of multimedia applications. We present an overview of the scheduling algorithm, describe its implementation in a commercial operating system, and discuss directions for future work.

1 Introduction

We are at the dawn of an era in which many everyday applications will need high performance computing. Tomorrow's multimedia applications will do much more than just playback pre-recorded audio and video; we expect that such applications will employ sophisticated image processing techniques and integrate complex computer graphics, all delivered with high interactivity and real-time response. To meet these computing demands, hardware platforms are evolving from single processor systems to multiprocessor systems. We already see today a proliferation of commercial multiprocessor solutions, from desktop workstations to high-end servers. By ganging together commodity microprocessors in varying multiprocessor configurations, these machines can leverage continuing improvements in microprocessor technology and offer the promise of scalable performance. The inherent parallelism found in many multimedia applications makes them particularly amenable to multiprocessor solutions.

While hardware technology has advanced, software technology has lagged behind in effectively supporting multimedia. A distinguishing characteristic of multimedia applications is that they often have real-time requirements associated with their execution. Recognizing the limitations of current operating systems in supporting the real-time requirements of multimedia applications, much work has been done to provide resource management mechanisms to address this problem [1, 3, 8, 9, 10, 11, 13, 15, 16, 18]. As processor cycles are often the most oversubscribed and critical resource, most of this work has focused on uniprocessor scheduling. However, hardware platforms are moving to multiprocessor systems and little work has been done to address the software problem of supporting multimedia applications in a multiprocessor context. Commercial operating systems such as UNIX SVR4 [17] and Windows NT [2] attempt to address this problem by providing a real-time static priority scheduler to be used in conjunction with a traditional timesharing scheduler, but experimental results have demonstrated that these approaches can result in pathological behaviors in which runaway real-time activities can take over the system and even prevent it from accepting user input [14].

To support the requirements of multimedia applications in a multiprocessor environment, operating systems must not only make efficient use of multiple processors in meeting the real-time requirements of these applications, but they must allow such real-time activities to co-exist with non-real-time (conventional) activities already found in traditional interactive and batch applications. While some previous work has attempted to address the problem of real-time multiprocessor scheduling [4, 6, 7], little work has been done to address the problem of how to allow both real-time and conventional applications to share resources and co-exist together in a multiprocessor environment. In addition, different users may have different preferences for the behavior of a particular application mix. To allow users to bias the allocation of resources according to those preferences, the operating system must also provide flexible user controls over the allocation of resources across both real-time and conventional applications.

Scheduling multimedia applications on multiprocessors poses challenges that do not arise in scheduling single processor systems. A single dispatch queue from which tasks are scheduled is sufficient for the uniprocessor case. However, experience with commercial operating systems suggests that scheduling multiple processors with a centralized dispatch queue is a synchronization bottleneck that can limit the scalability of multiprocessor systems [5, 19]. If a dispatch queue should be associated with each processor, how should tasks be assigned to those dispatch queues in the first place? In particular, the operating system must effectively balance the load across multiple processors. As multimedia application workloads often have dynamically varying resource demands, the operating system must decide when to migrate tasks from one processor to another for load balancing. On the other hand, the operating system may want to reduce task migration through some form of cache affinity to reduce cache misses that occur when a task migrates among processors.

This paper proposes a new multiprocessor scheduler designed to meet the requirements of multimedia applications. Some of the key features of our solution are: (1) decouples the assignment of which processor to use to run a given task (processor task assignment) from the scheduling of tasks already assigned to a processor (per processor scheduling), (2) accounts for cache effects in the processor task assignment, (3) explicitly accounts for application time constraints in both the processor task assignment and the per processor scheduling to make efficient use of resources in meeting real-time requirements, (4) provides flexible prioritized and proportional resource sharing across both real-time and conventional activities. We describe the usage model of our system, present an overview of the scheduling algorithm, and discuss its implementation in the Solaris UNIX operating system, Sun's commercial operating system.

2 Usage model

Our scheduler provides explicit time constraints for real-time applications and supports the notions of priority and proportional sharing across real-time and conventional tasks, even when the system is overloaded. Without additional information from the user, the system provides fair resource allocation across all tasks by default. The system can deliver a rich set of behavior with just a little more information from users and applications:

- **Time constraints**. An application can provide time constraints on a computation, and the scheduler will try to schedule earlier deadline computations before computations that can tolerate more delay. A time constraint for a block of application code consists of a deadline by which the code should execute and an estimate of the processing time required for the code.
- **Resource allocation: priority and proportional sharing**. Real-time tasks should not always be allowed to run first because they may starve out important conventional tasks, such as those that keep the system running. The user can prioritize all the tasks as desired, regardless of whether they are real-time or conventional. Each task can be assigned a priority for this purpose. Among tasks of equal priority, the user can also decide that certain applications should share a processor in some given proportion, which is maintained adaptively as the loading condition changes. Each task can be assigned a share for this purpose.
- **Inform applications of resource availability**. In support of adaptive real-time applications, our scheduler allows applications to decide their own degradation policies. The scheduler notifies applications via an upcall when their time constraints cannot be met and informs them of what resources are available. The upcall invokes an application-level handler which can decide its own policy for dealing with the missed time constraint. For instance, a video application can decide whether to skip a video frame or show a lower quality image when the frame cannot be fully displayed in a timely fashion.

In summary, each task can be parameterized based on its time constraint, consisting of a deadline and a processing time estimate, and its priority and share, which determine the overall resource allocation of the task.

3 Scheduling algorithm

The approach that we take in our scheduler design is to reduce the multiprocessor scheduling problem into two decoupled scheduling decisions: (1) selecting the processor to which to assign a set of tasks, and (2) scheduling the set of tasks assigned to a given processor. By decoupling these decisions, each processor can schedule its own set of tasks in an independent manner. This distributed scheduling model avoids excessive synchronization with other processors and provides performance that can scale with multiprocessor machines with larger numbers of processors.

The basic idea behind how our scheduler decides which processor to assign a set of tasks is as follows. Given a real-time task, find the best processor to use to execute the given task such that all real-time tasks assigned to that processor can meet their respective deadlines. As a result, the algorithm finds excess slack in the system and uses it effectively for meeting deadlines. If no such processor exists, assign the task in question to the processor with the least important tasks, where importance is measured based on the priorities and shares of tasks. As a result, the algorithm finds processors that are executing less important tasks and defers them to ensure that more important tasks can obtain their desired resource allocations.

In selecting a processor to which to assign a real-time task, the scheduling algorithm determines whether the given real-time task can be run on the processor such that all real-time tasks on that processor can meet their deadlines. If more than one processor satisfies this criterion, the algorithm selects the processor whose excess slack is the best fit for the computational requirements of the given real-time task. This packing process allows real-time tasks with less stringent time constraints to share a given processor while ensuring that processors with large amounts of slack remain available for running real-time tasks with more stringent time constraints. In particular, a processor with conventional tasks assigned to it is considered to have zero excess slack. As a result, real-time tasks will tend to be assigned to processors with other real-time tasks and no conventional tasks. Since real-time tasks, such as those that process video, have poor cache behavior, their execution on the same processor does not result in caching problems. At the same time, they also do not cause cache pollution on processors that are executing conventional tasks with good cache behavior.

In selecting a processor to which to assign a task based on importance, the scheduling algorithm uses priority as the primary measure of importance. It calculates the processor priority for each processor based on the highest priority task assigned to the processor, and selects the processor with the lowest processor priority. For processors of the same priority, the scheduling algorithm sums the total number of shares of all tasks at the given priority and selects the processor with the smallest share sum.

Note that in selecting a processor to which to assign a conventional task, the scheduling algorithm does not use the slack of each processor. Instead, it simply uses the importance-based selection. Once a conventional task is assigned to a given processor, the algorithm may migrate the task to another processor for load balancing reasons, but will avoid doing so for a predefined interval of time. This helps keep the working set of the task in the respective cache and reduce cache misses that result from switching processors. Cache affinity considerations are less of an issue for real-time tasks since they exhibit poor cache behavior.

Once tasks have been assigned to a given processor, we use the SMART scheduling algorithm [15] to perform the per processor scheduling because of its demonstrated performance benefits over other uniprocessor approaches. The algorithm examines both real-time and conventional tasks and performs a feasibility test to determine whether or not the deadlines of all real-time tasks can be met while ensuring that conventional tasks receive their desired resource allocations. When not all deadlines can be met, less important tasks are deferred to meet the requirements of more important tasks, where importance is measured based on the priorities and shares of tasks. A detailed discussion of the SMART scheduling algorithm is beyond the scope of this paper, but such a discussion presented in [15].

4 Implementation status and future work

We have implemented our scheduling algorithm in version 2.5.1 of Sun's Solaris UNIX operating system, a commercial UNIX SVR4 multithreaded operating system. The Solaris operating system is already designed to run on a shared-memory multiprocessor system, which aided our implementation effort. In particular, the Solaris scheduling framework provides a separate dispatch queue for each processor and each task is assigned to one dispatch queue from which it can execute. Separate scheduling locks are associated with each dispatch queue. Scheduling operations proceed in a sequence of steps, obtaining and releasing necessary scheduling locks. Such incremental steps allow scheduling operations to proceed in parallel on different processors, providing more scalable performance.

The original UNIX SVR4 scheduler is a two-level scheduler with a set of scheduling class policies and an underlying priority scheduler. The scheduling class policies include a real-time static priority class, a system priority class, and a timesharing class. The various scheduling policies are unified by mapping each of them onto a range of global priorities. The global priorities are then used by the underlying priority scheduler to decide which task to execute.

To implement our scheduling algorithm, we replaced the underlying UNIX SVR4 priority scheduler with our scheduling algorithm. In particular, our decoupling of processor task assignment and per processor scheduling mapped well to the existing Solaris scheduling framework of per processor dispatch queues. We also introduced a new scheduling policy that exposes the features of our scheduling algorithm to applications and users to allow them to take advantage of time constraints, priorities, and shares. One of the benefits of our approach is that our scheduler subsumes basic priority scheduling and is thus completely backwards compatible with standard UNIX SVR4, allowing all scheduling class policies to preserve their existing functionality without any modification.

Some modifications were also made to the UNIX SVR4 scheduling framework to enable the system to more precisely support the timing requirements of multimedia applications. In particular, the standard UNIX SVR4 scheduling framework, upon which the Solaris operating system is based, employs a periodic 10 ms clock tick. It is at this granularity that scheduling events can occur, which can be quite limiting in supporting real-time computations with audio and video that have time constraints of the same order of magnitude. To allow a much finer resolution for scheduling events, we added a high resolution timeout mechanism to the kernel and reduced the time scale at which timer based interrupts can occur. The exact resolution allowed is hardware dependent, but is typically 1 ms or less.

Our initial experiences in running mixes of multimedia audio and video applications with our prototype scheduler implementation have shown promising qualitative improvements over standard UNIX SVR4 scheduling on Sun multiprocessor configurations of up to six processors. While effective multiprocessor scheduling is crucial to support multimedia applications, the processors are just one set of components in an overall system. Other resources that require effective resource management include I/O bandwidth, memory, and the network and network/host interface. Meeting the demands of future multiprocessor multimedia applications will require coordinated resource management across all critical resources in the system. We believe that the ideas discussed here for multiprocessor scheduling will serve as a basis for future work in addressing the larger problem of managing system-wide resources to support multimedia applications.

References

1. G. Bollella, K. Jeffay, "Support for Real-Time Computing Within General Purpose Operating Systems: Supporting Co-Resident Operating Systems", *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Chicago, IL, pp. 4-14, May 1995.
2. H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.
3. A. Demers, S. Keshav, S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm", *Proceedings of SIGCOMM '89*, pp. 1-12, Sept. 1989.
4. M. Dertouzos, A. Mok, "Multiprocessor On-line Scheduling of Hard-Real-Time Tasks", *IEEE Transactions on Software Engineering*, 15(12), pp. 1497-1506, Dec. 1989.
5. J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams, "Beyond Multiprocessing...Multithreading the SunOS Kernel", *Proceedings of the 1992 Summer USENIX Conference*, San Antonio, TX, pp. 11-18, June 1992.
6. H. Forbes, K. Schwan, "Rapid – A Multiprocessor Scheduler for Dynamic Real-Time Applications", Technical Report GIT-CC-94-23, College of Computing, Georgia Institute of Technology, Apr. 1994.
7. M. R. Garey, D. S. Johnson, "Scheduling Tasks with Nonuniform Deadlines on Two Processors", *JACM*, 23(3), pp. 461-467, July 1976.
8. D. B. Golub, "Operating System Support for Coexistence of Real-Time and Conventional Scheduling", Technical Report CMU-CS-94-212, School of Computer Science, Carnegie Mellon University, Nov. 1994.
9. P. Goyal, X. Guo, H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems", *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, pp. 107-122, Oct. 1996.
10. M. B. Jones, D. Rosu, M-C. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, pp. 198-211, Oct. 1997.
11. I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", *IEEE JSAC*, 14(7), pp. 1280-1297, Sept. 1996.
12. C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *JACM*, 20(1), pp. 46-61, Jan. 1973.
13. C. W. Mercer, S. Savage, H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications", *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, pp. 90-99, May 1994.
14. J. Nieh, J. G. Hanko, J. D. Northcutt, G. A. Wall, "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications", *Proceedings of the Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Nov. 1993.
15. J. Nieh, M. S. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications", *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, pp. 184-197, Oct. 1997.
16. I. Stoica, H. Abdel-Wahab, K. Jeffay, "On the Duality between Resource Reservation and Proportional Share Resource Allocation", *Multimedia Computing and Networking Proceedings, SPIE Proceedings Series*, Vol. 3020, San Jose, CA, pp. 207-214, Feb. 1997.
17. *UNIX System V Release 4 Internals Student Guide*, Vol. I, Unit 2.4.2., AT&T, 1990.
18. C. A. Waldspurger, "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management", Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.
19. J. Zolnowsky, "Realtime Dispatch in SunOS: an Update", *SunSoft TechConf '96*, Apr. 1996.