Scheduling and Resource Management Techniques for Multiprocessors

David L. Black

July 1990

CMU-CS-90-152

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to Carnegie Mellon University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, ARPA Order No. 5993 .

Keywords: Multiprocessor, Operating System, Mach, Scheduling, Handoff Scheduling, Gang Scheduling, Discouragement, NUMA, Memory Management, Competitive Algorithms, Timing, Parallelism, Concurrency

Abstract

This thesis describes research in the design and implementation of multiprocessor resource management and related areas. Application requirements motivated the major research areas, processor scheduling and non-uniform memory management, as these areas contain the most important problems raised by the changing design and use of multiprocessors.

A variety of processor scheduling problems are addressed by this work. A major contribution is the design and implementation of a processor allocation and control facility for the Mach operating system that allows processors to be dedicated to applications. The implementation of this facility has been validated by production use, and it is one of the first successful applications of policy-mechanism separation to multiprocessor scheduling (and, in particular, to the problem of dividing a multiprocessor among competing applications). The techniques of Discouragement and Handoff Scheduling are introduced to address the communication and synchronization problems of applications with more active entities than processors. Support for timesharing has been preserved by integrating the new features into the Mach timesharing scheduler. In the process, it was necessary to redesign the scheduler from scratch, and a number of lessons were learned about the design and implementation of timesharing schedulers, including how to collect accurate processor usage information based on timestamps instead of interrupt-driven sampling.

The non-uniform memory management research takes a different approach, predicated on obtaining more information from hardware. Current implementations of non-uniform memory management are restricted by the available reference information. Hardware reference and modification bits indicate only that a page was referenced or modified at least once sometime in the past; they do not indicate how many references were made or how long ago. If more reference information were available, more sophisticated (and hopefully better) management techniques would be possible. This research assumes that any desired information about the access history of any region of memory is available and designs competitive replication and migration algorithms that can take advantage of this information. A case study examines how these algorithms could be incorporated into the Mach operating system.

To my parents
and grandparents

Acknowledgments

I'd like to express my gratitude to the many people who helped make this research and thesis possible. First and foremost to my advisor, Rick Rashid, whose support, encouragement, and enthusiasm were instrumental in seeing this work through to its completion. Thanks also to the other members of my thesis committee, Hide Tokuda, Zary Segall, and Ike Nassi, for their support and continued interest in my work. Danny Sleator deserves a special note of gratitude for his generous assistance with development of the competitive algorithms, and his diligent checking of details in the associated proofs.

The School of Computer Science at Carnegie Mellon and the Mach project provided a supportive environment, allowing me the freedom to concentrate on research issues. I owe a debt of gratitude to Avie Tevanian for explaining some of the intricacies of Mach and answering my questions in the early days when we shared an office. Other members of the Mach project helped in many ways. My thanks to Bob Baron, Mike Young, David Golub, Bill Bolosky, Richard Draves, Alessandro Forin, Joe Barrera, Charles Hill, Mike Jones, Jonathan Chew, Rich Sanzi, and Mary Thompson. Serious research on large software systems is heavily dependent on the combined efforts of talented people.

Some of this work was furthered by cooperation with researchers at Stanford University. I'd like to thank Anoop Gupta, Wolf-Dietrich Weber, and Andy Tucker for their contributions to these efforts. Additional inspiration came from design discussions with researchers at IBM Research's Hawthorne facility. Thanks in particular to Ray Bryant, Bryan Rosenburg, and Henry Chang, for their participation and interest. I'd also like to thank Gregg Kellogg of NeXT for reviewing and commenting on preliminary versions of some of the interfaces.

Most of the implementation work was performed on an Encore Multimax. Many thanks to the people at Encore who helped port and support Mach on the various versions of the Multimax, including Jim Van Sciver, Fred Oliveira, David Kirschen, Peter Fay, Joseph Boykin, Alan Langerman, and Susan LoVerso.

Feedback from users was important in improving the design and implementation of the Mach scheduler. My thanks to all the users of the Multimax who coped with software under development and provided useful bug reports, including Dirk Kalp, Wilson Harvey, Milind Tambe, eddie caplan, and Ted Lehr. Also thanks to Allan Fisher, Thomas Gross, and the students in their parallel programming courses for putting the processor allocation software through its paces.

Sharon Burks was ever-vigilant in protecting graduate students from the rest of the University. Sid Chatterjee helped explain the intricacies of LaTeX, and assisted with development of the processor allocation server at a crucial juncture. Many thanks to Catherine Copetas and all of the other people who make Carnegie Mellon such an interesting and stimulating place to do research.

And last but not least, my wife, Kathryn, somehow convinced LaTeX to produce most of the diagrams. Her patient proofreading and copy-editing of several drafts have greatly improved this document. I am truly grateful, and any errors that may remain are mine alone.

Live long and prosper. Pittsburgh, July 26, 1990.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes research in the design and implementation of multiprocessor resource management and related areas. New approaches to resource management are required by significant changes in multiprocessor architectures and programming. The use of processing resources is changed by techniques such as parallel programming and multithreading that employ more than one independently executing entity (e.g., process, thread) in a single application. This cooperation within applications conflicts with the usual timesharing notion that all schedulable entities compete independently. A recent trend in large multiprocessors is the use of non-uniform memory, so that different processors have different access times to the same region of memory. Such designs invalidate resource management techniques based on uniform memory, because the memory can no longer be treated as a single large pool. This research investigates new techniques for resource management; its primary focus is on the operating system, but important consideration is given to utilizing information that may be available from applications. The changing requirements of applications motivate the major research areas of processor scheduling and non-uniform memory management because they contain the most important problems raised by the changing design and use of multiprocessors.

The collection of problems involved in managing processing resources, or *scheduling*, can be characterized by the class of applications that exhibits each problem. Some parallel applications require dedicated use of processors for acceptable performance, and cannot tolerate sharing these with any other uses (as would be the case under timesharing). This is compounded by the variety of languages and programming models that exhibit such requirements. A second class of applications uses more active entities than processors and needs support for effective communication and synchronization among them. A third problem occurs in integrating support for these classes of applications without losing sight of timesharing, which is still an important (if not dominant) class of applications for multiprocessors.

This research employs a number of techniques to address these scheduling problems. Extensions and modifications to an existing scheduling methodology are emphasized to preserve support for timesharing. A major contribution of this work is the design and implementation of a processor allocation and control facility for the Mach operating system[42] to allow processors to be dedicated to applications. The implementation of this facility has been validated by production use, and it is one of the first successful applications of policy-mechanism separation to multiprocessor scheduling (and in particular, to the problem of dividing a multiprocessor among competing applications). The techniques of *Discouragement and Handoff Scheduling* are introduced to address the communication and synchronization problems of applications with more active entities than processors. These techniques yield significant performance improvements for communication and synchronization in this class of applications. Support for timesharing has been preserved by integrating the new features into the Mach timesharing scheduler. In the process, it was necessary to redesign the scheduler from scratch, and a number of lessons were learned about the design and implementation of timesharing schedulers. An important contribution in this area is the design and implementation of a timestamp-based timing facility to measure processor usage on both uniprocessors and multiprocessors. This timing facility eliminates the distortion in measurements obtained from current statistical timing implementations and improves accuracy by multiple orders of magnitude at a minimal overhead cost. A modicum of hardware support is required to generate the timestamps.

The non-uniform memory management research takes a different approach, predicated on obtaining more information from hardware. Current implementations of non-uniform memory management are restricted by the available reference information. Hardware reference and modification bits indicate only that a page was referenced or modified at least once sometime in the past; they do not indicate how many references were made or how long ago. If more reference information were available, more sophisticated (and hopefully better) management techniques would be possible. This research assumes that any desired information about the access history of any region of memory is available and investigates replication and migration algorithms that can take advantage of this information. These algorithms cannot be implemented in the current generation of non-uniform memory access (NUMA) multiprocessors, but may suggest appropriate hardware support for future designs.

The major contributions of the non-uniform memory management research are new competitive algorithms and a case study of how they could be incorporated into Mach. The primary contribution is new on-line algorithms for several important cases of the replication and migration problems. The performance of these algorithms is within a small constant factor (2–4) of optimal; this property of on-line algorithms is called *competitiveness*. The migration algorithms are an important theoretical contribution, as they are the first competitive algorithms for the *1-server with excursions* problem. This research includes not only the design and (competitive) analysis of these algorithms, but also a design study of how they could be incorporated into Mach. The chosen cases of the replication and migration problems apply to most current and proposed NUMA multiprocessor designs.

1.1 Programming Models and Scheduling Requirements

The scheduling research is motivated by the scheduling requirements of parallel and concurrent programs, which are derived from a classification of their programming models. The classification is based on the additional concurrency (if any) that the models introduce beyond the basic hardware parallelism. An operating system can introduce concurrency by providing independently schedulable entities (e.g., Unix processes, Mach threads) that it multiplexes onto the hardware processors. These entities are called *virtual processors* or VPs. A user level library or language runtime can introduce further concurrency by multiplexing language level entities onto these VPs. These language entities are called *multiroutines*; they may be thought of as multiprocessor generalizations of coroutines. Specific examples of multiroutines include coroutines (only 1 VP), and the usual programming notion of multiple threads (1 VP per multiroutine). Four classes emerge when these models are classified by their use of concurrency, ranging from *pure parallelism* models that use no concurrency beyond the hardware parallelism to *dual concurrent* models that take advantage of concurrency at both the operating system and user levels.

Scheduling support requirements for parallel and concurrent programs can be identified based on this classification. Two such requirements emerge:

1. Models that assume the availability of processors for parallel execution require guarantees of that availability.

2. Models that employ system concurrency (e.g., more threads than processors) require support to manage this concurrency (especially for efficient synchronization and communication).

The processor allocation and control mechanisms address the first area of support, and the second is addressed by the techniques of Discouragement and Handoff Scheduling.

1.2 Processor Allocation and Control

This thesis describes the design and implementation of a processor allocation and control facility for the Mach operating system. This facility allows the provision of dedicated processors to applications that need them and allows such applications to control their use. By separating policy and mechanism, this facility can divide processor resources among competing applications without involving the operating system kernel in the details of policy, programming models, or multiprocessor topology. The operating system kernel implements the basic mechanisms for processor allocation and control. Allocation of resources among competing applications is handled by a privileged server which encapsulates most knowledge about policy and topology. Allocation of resources within a single application is handled by the application itself; this encapsulates most knowledge about programming models within the individual applications. The overall design of the facility allows applications to manage resources without giving them absolute control; the server can always remove resources from a recalcitrant application. Changing the allocation policy requires replacing, at most, the server and application interfaces to it; the kernel (which is much harder to replace) is not affected.

A processor allocation server for a simple policy and the interfaces to it have been implemented to exercise and validate the processor allocation and control facilities. This server implements a greedy FCFS (first come, first served) processor allocation policy subject to fixed time and resource limits. The policy means that as many requests as there are processors for will be granted, but the server will not reorder the request queue (e.g., move a smaller request ahead of a larger one). The resulting server bears distinct similarities to a batch scheduler for physical memory. The server has been successfully used for research and for teaching an undergraduate parallel programming course at Carnegie Mellon. The latter use allowed undergraduates to allocate processors when needed for generation of speedup curves without introducing administrative difficulties, or even the need for recompilation of the measured applications.

Many extensions and changes to the policy implemented in this server are possible. Since it is a batch scheduler for processors, techniques originally developed for batch scheduling of memory, such as assigning higher priority to shorter requests, are applicable. In addition, the server could be extended to allow some users higher or absolute priority in allocating processors, or to allow more processors to be allocated during light usage periods. Finally, the server can be replaced in its entirety by a server that implements a different scheduling policy. One promising new policy is to vary the number of processors available to applications based on the overall demand for processors. A server with this policy can notify applications to reconfigure when it changes the number of processors available. Researchers at Stanford are pursuing this approach and have implemented a server for this scheduling policy under Mach with good initial results[51].

1.3 Discouragement and Handoff Scheduling

Programs that use more virtual processors (e.g., Mach threads) than physical processors need support for managing the resulting system concurrency. An important problem faced by such programs is interacting (communicating or synchronizing) with a virtual processor that isn't currently executing on a physical processor. In the absence of specific support, the best that can be done is to block the virtual processor, or wait for its time slice to run out in the hopes that the scheduler will run the desired virtual processor. Both of these approaches entail potentially prohibitive overheads (e.g., blocking on a Unix system can require two system calls that use hundreds of instructions each). This thesis describes the design and implementation of two better approaches, Discouragement and Handoff scheduling.

The overall design implements the scheduler in the kernel, but allows users to provide hints based on local information. Implementing the scheduler in the kernel avoids the expense of crossing the user-kernel boundary to make critical scheduler decisions, and also avoids the need for users to maintain complex information about the overall state of the application (two major drawbacks of some past efforts in this area[57, 25]). Local hints take advantage of scheduling information that is readily available to users; the information that the current thread cannot make progress, and the identity of some other thread that should be run instead. For example, a thread may be unable to acquire a lock, or it may encounter an empty buffer and know the identity of the (unique) thread that fills that buffer. A local hint only affects the thread that provides it, and possibly one other thread. The use of hints rather than directives is important because it allows the information to be wrong. In the buffer example, the thread that fills the buffer may not be runnable because it is blocked waiting for something else; the kernel can check this more effectively than the application.

Discouragement scheduling involves hints that the current thread should not be run. The mildest hint surrenders the thread's time slice to any other thread that can use it. Stronger hints discourage the scheduler from assigning another time slice by depressing the thread's priority or blocking it for a time period. These hints are intended for use when it is not known which thread is being waited for (e.g., some other thread holds a lock, but the lock doesn't record the identity of the holder). The range of hints is motivated by priority inversion and timing granularity problems encountered in building real schedulers.

Handoff scheduling employs hints that explicitly specify the thread that should be run instead of the current thread. If that thread is runnable but not currently running, the processor will be directly "handed off" to it, bypassing the scheduler's normal thread selection algorithm. This yields two advantages:

1. The desired thread runs immediately if possible.
2. The context switch is more efficient because the code path through the scheduler is shorter.

Uses in the operating system kernel gain additional efficiency by optimizing wakeup and handoff operations in a single path. A handoff hint always implies some form of discouragement hint, because the thread making the handoff request is indicating a willingness to give up the processor. A single kernel trap can provide both hints to allow any form of discouragement to be coupled with handoff scheduling.

An experimental evaluation of Discouragement and Handoff Scheduling demonstrates their usefulness. Synchronization experiments show that both Discouragement and Handoff hints improve performance of synchronization in a program using more threads than processors over a simple context switch strategy. Handoff hints have much better performance, and are faster than sending a message. The use of Handoff Scheduling within the kernel also improves the performance of remote procedure calls by shortening the code path required to run the receiver. This result holds for a multiprocessor because a handoff can be used to achieve faster dispatch of the recipient on an idle processor. No performance improvement was observed for the unidirectional message case because the sender can queue multiple messages per context switch in the uniprocessor case, and the sender runs in parallel with the receiver on a multiprocessor.

1.4 Scheduler Design and Implementation

The redesign and reimplementation of the Mach scheduler (necessitated by the need to preserve timesharing functionality and incorporate the new features described above) pioneered the use of several techniques for implementing timesharing schedulers and related facilities. The scheduler avoids the need for periodic scans of all threads in the system by making individual threads responsible for collecting and updating their own usage information. A periodic scan of all threads on the run queues is still necessary to avoid starvation, but the number of threads involved is much smaller. The scheduler employs a new usage aging algorithm that avoids the use of floating point numbers and division operations in the scheduler. Accurate cpu usage information is collected for both users and the scheduler by a new timing subsystem based on timestamps from a hardware timestamp source (a free running counter at a known clock frequency is sufficient). This avoids the distortions caused by statistical timing, improves timing accuracy by several orders of magnitude, and bases scheduler decisions on these accurate usage times instead of potentially inaccurate approximations. A mutual exclusion protocol based on techniques developed by Lamport[30, 31] is used to avoid interlocked instructions in this subsystem since it is invoked from the critical trap, interrupt, and context switch paths in the kernel. A caching technique is used to make the resulting usage information efficiently available to the scheduler.

1.5 NUMA Memory Management

The research on NUMA memory management is motivated by changes in multiprocessor architectures. Designers of large multiprocessors are moving away from uniform memory access (UMA) architectures to non-uniform memory access (NUMA) architectures. In the memory system, this entails a change from a structure in which all memory is equally close to all processors (in terms of access time) to one in which some memory is close to each processor, but different processors have different access times to the same memory. The latter (NUMA) structure is referred to as *physically distributed* memory because the memory is physically distributed throughout the machine to make sure each processor is close to some memory. New techniques are required to manage these memory structures because the location of memory used by an application (with respect to the processors executing the application) directly affects performance. Optimizing the use of physical memory to minimize access costs is a major concern for both applications and operating systems on such architectures.

New on-line competitive algorithms for managing memory in NUMA architectures have been developed as a result of this research. An algorithm is *on-line* if it must respond to a sequence of requests without advance knowledge of future requests (elements of the sequence). An on-line algorithm is *competitive* if the costs it incurs for any sequence are within a small constant factor of the smallest possible costs for that sequence (i.e., those costs incurred by the best algorithm that has advance knowledge of the entire sequence). A *strongly competitive* algorithm is one for which the constant factor is the smallest possible. For this problem, the 'costs' involved are the extra costs of accessing remote memories, plus any costs incurred in copying data among memories. An informal statement of the competitive properties of these algorithms is that their performance is the best that can be achieved in the absence of knowledge about the future, and knowledge of the entire future can only improve performance by the small constant factor.

This thesis presents and analyzes new competitive and strongly competitive algorithms for two important problems that occur in NUMA memory management, *replication* and *migration*. We consider topologies that correspond to common NUMA multiprocessor interconnection structures, the complete graph (corresponding to both bus and complete network interconnects), and trees. The migration problem solutions also extend naturally to product graphs, which correspond to hypercubes and meshes. The replication results are an extension of well-known results in the field, but the migration results are significant new contributions to this area of theory; they are the first competitive and strongly competitive algorithms (as well as a lower bound result) for the *1-server with excursions* problem. The potential use of these algorithms in an operating system is also discussed, using Mach as a case study.

1.6 Structure of the Thesis

To provide context and background for the scheduling portion of this thesis, Chapter 2 begins with a discussion of the Mach scheduler and the techniques used to implement timesharing. Chapter 3 continues this topic with a description of the accurate usage timing system used to gather usage information for both the scheduler and users. Performance measurements are presented to demonstrate the low cost of obtaining this information.

Programming models and classification of parallel and concurrent programs is covered in Chapter 4, which also identifies the new scheduling requirements of these programs. The next two chapters cover the new Mach features that address these requirements. Chapter 5 covers Discouragement and Handoff Scheduling and its support for concurrent programs. Chapter 6 covers the Processor Allocation and Control subsystem and its support for parallel programs.

NUMA memory management problems are considered in the next two chapters. Chapter 7 formally defines the replication and migration problems for NUMA architectures, introduces new algorithms to solve them, and provides proofs of their competitive properties. Chapter 8 considers some of the important issues that arise in actually implementing these algorithms in real systems, using Mach as a case study.

The thesis concludes with Chapter 9. Five appendices provide more detailed information on the Mach features discussed throughout the thesis.

Chapter 2

Timesharing and the Mach Scheduler

To provide a basis for the investigation of multiprocessor scheduling, this chapter covers timesharing scheduling and describes the techniques used in the Mach scheduler. An historic overview of timesharing scheduling describes the major techniques used for timesharing and some of the problems encountered in the past. One important conclusion is that a timesharing scheduler based on usage measurements must take the system load into account to produce reasonable behavior under high load. This overview is followed by a description of the internals of the Mach scheduler including the state machine used to manage individual threads and the techniques used to adjust priorities of threads based on processor usage. An important contribution in the area of priority management is the use of approximations based on shifts and adds to replace and optimize priority manipulations that would otherwise require multiplication and division instructions. The chapter concludes with descriptions of techniques used to reduce the number of context switches for load and usage balancing on both uniprocessors and multiprocessors.

2.1 Timesharing Scheduling

A major goal of timesharing schedulers is fair sharing of processor resources so that applications competing for processor time will receive approximately equal portions. The notion of 'approximately equal' applies over fairly short periods of time (seconds) to ensure interactive response in the presence of compute-bound jobs. In practice, this sharing requires some form of processor usage tracking and the use of this information in scheduling decisions. The simplest form of this is a decreasing priority scheduler in which the priority of a process decreases continuously as it uses processor time, and the scheduler favors higher priority processes. Multics used such a scheduler and discovered its major disadvantage; on a heavily loaded system with significant numbers of short-lived jobs, the priority of a lengthy job can decrease to the point that little or no further processor time is available to it [39]. The automatic depression of priority for lengthy jobs in some versions of Unix also exhibits this drawback.

To avoid the problem of permanently depressed priorities, it is necessary to elevate them in some manner. There are two major choices for elevation methodologies; event-based elevation, and processor usage aging. Event-based elevation is used to deliberately favor interactive response over compute-bound jobs because process priority is elevated by the occurrence of events such as I/O completion. The elevations associated with the events of interest must be determined in some manner, such as tuning under expected workloads to produce the desired response behavior. This methodology assumes that jobs are either distinctly compute-bound or distinctly interactive and that interactive jobs are more important. Users whose interactive work consumes large amounts of processor time may not do well under this methodology. The VAX/VMS[1] scheduler employs this elevation methodology [29], as does the University of Maryland's ESCHED scheduler for large Unix systems [46].

The second priority elevation methodology is processor usage aging. Its original form is based on expected usage functions that define the goals for processor usage based on the time a job has been in the system (its *elapsed time*)[2, 43]. Each job's priority is based on how close its actual usage comes to this goal. To reduce overhead, the goal function is inverted to calculate the time by which a job's current processor usage should have been attained (the job's *critical time*). Each job's priority is the difference between its elapsed time and its critical time. The priority of a job that receives no processor time is elevated as this difference increases. Multiple policy functions can be used to provide different levels of service to distinct classes of jobs. The definition of the term 'job' is important to an understanding of these schedulers; they are intended for environments composed of (a limited number of) batch jobs and distinct (short-lived) interactions with users. A user's terminal session would be composed of a large number of interactions (e.g., one for each command given to the system). Long-lived interactions pose problems for these schedulers because the expected usage function is not sensitive to load. An expected usage function can be tuned to a single load, but then variations in load produce unacceptable behavior. If the load rises above the tuned level, old interactions are favored over new ones and system responsiveness suffers; if the load falls, new interactions are favored over old ones leading to the same starvation problem found in Multics. Long-lived interactions are common on current systems, making this class of schedulers inappropriate.

Modern processor usage aging schedulers elevate priorities by gradually forgetting about past processor usage, usually in an exponential fashion. An example is that usage from one minute ago is half as costly as usage within the past minute, usage from the minute before is half again as costly, and so on. As a result, the scheduler's measure of processor usage is an exponentially weighted average over the lifetime of a process. A simple exponential average is not desirable because it has the unexpected side effect of raising priorities when the load on the system rises. This happens because under higher loads each process gets a smaller share of the processor, so its usage average drops, causing its priority to rise. These elevated priorities can degrade system response under heavy loads because no process accumulates usage fast enough to drop its priority. The 4.3BSD version of Unix solves this problem by making the aging rate depend on the load factor so that aging is slower in the presence of higher load, keeping priorities in more or less the same range [34]. An alternative technique is to use an overload factor to alter the rate at which usage is accumulated. For this technique, the usage accumulated by the scheduler is the actual usage multiplied by a factor that reflects the increased load on the system. Although these schedulers are a development of the older expected usage function schedulers, the expected usage functions have vanished because priority is calculated by comparing usage among processes instead of against individual goals.

For a multiprocessor scheduler, the concept of fair sharing is strongly intertwined with that of load balancing. The goal of load balancing is to spread the system's computational load evenly among the available processors over time. For example, if three similar jobs are running on a two processor system, each should get two-thirds of a processor (on average). This often requires the scheduler to shuffle processes among processors to keep the load balanced. There is an important tradeoff between load balancing and overhead in that optimal load balancing causes high scheduler overhead due to frequent context switches. A uniprocessor timesharing scheduler encounters similar

[1] VAX/VMS is a trademark of Digital Equipment Corporation.

issues in minimizing the number of context switches used to implement fair sharing.

2.2 The Mach Scheduler

Although Mach splits the traditional process notion into the task (address space and communication capabilities) and thread (locus of control, program counter, and registers) abstractions, the Mach scheduler only schedules threads. The knowledge that two threads are in the same task can be used to optimize the context switch between them. This information is not used to select the threads to run because there may be little improvement in performance depending on the hardware and application involved, and because favoring threads on this basis can be detrimental to load and usage balancing. Some scheduler functions are driven by periodic clock or timer interrupts that occur on all processors every 10 to 100 milliseconds. Mach uses the same scheduler for both multiprocessors and uniprocessors; this description is oriented towards the multiprocessor use of the scheduler, but is also applicable to uniprocessors. The original Mach scheduler design (including the run queue structure) is due to Avadis Tevanian Jr., and the state machine design and implementation was done by David B. Golub.

The virtual memory system's swapping functionality interacts with thread scheduling. An important resource associated with each thread is its *kernel stack*, the stack used by the kernel to execute operations on the thread's behalf. This stack must be memory resident when the thread is executing because there is no other stack that can be used to handle a page fault on the memory that it occupies. Mach allows the kernel stacks of inactive threads to be *swapped* to free up their memory resources. The action of swapping out a thread consists of making its stack memory eligible for the normal pageout mechanisms. Neither the stack nor any other memory is explicitly forced out of memory by this action. The corresponding action of swapping in a thread consists of forcing its kernel stack into memory, and *wiring* it so that it cannot be paged out.

2.2.1 Thread States

There are 14 possible scheduling states for a Mach thread. These and the transitions among them are shown in Figure 2.1; scheduler actions that do not cause state changes are omitted for clarity. This figure organizes the states into three nested squares (four states in the inner square, five in both the middle and outer squares). The components of state names correspond to the state of the thread; running (**R**), waiting (**W**), suspended (**S**), and swapped out (**O**). The suffix **proc** is added to states in which the thread is executing on a processor, and the suffix **runq** indicates that the thread is queued to run (on a run queue). The innermost square contains the four states in which a thread is executing on a processor, as indicated by the presence of **proc** in their names. The outermost square contains the five states in which a thread is swapped out; in this state the thread's kernel stack may not be resident in memory and must be forced into memory (swapin) before the thread can be executed. The middle square contains the five states in which a thread is swapped in (kernel stack is memory resident), but not actually running; such a thread may be waiting on an event, suspended by the kernel, or queued to run on a processor. Table 2.1 describes the states in the figure, and Table 2.2 describes the labeled state transitions. The second column in Table 2.2 gives the actual name of the internal Mach routine that performs the action; these names are abbreviated in Figure 2.1 for clarity. These routines are not part of the kernel interface exported by Mach.

Vertical transitions in Figure 2.1 involve the suspend mechanism. This is used by the kernel to prevent a thread from executing (e.g., a thread that is about to be terminated must be stopped

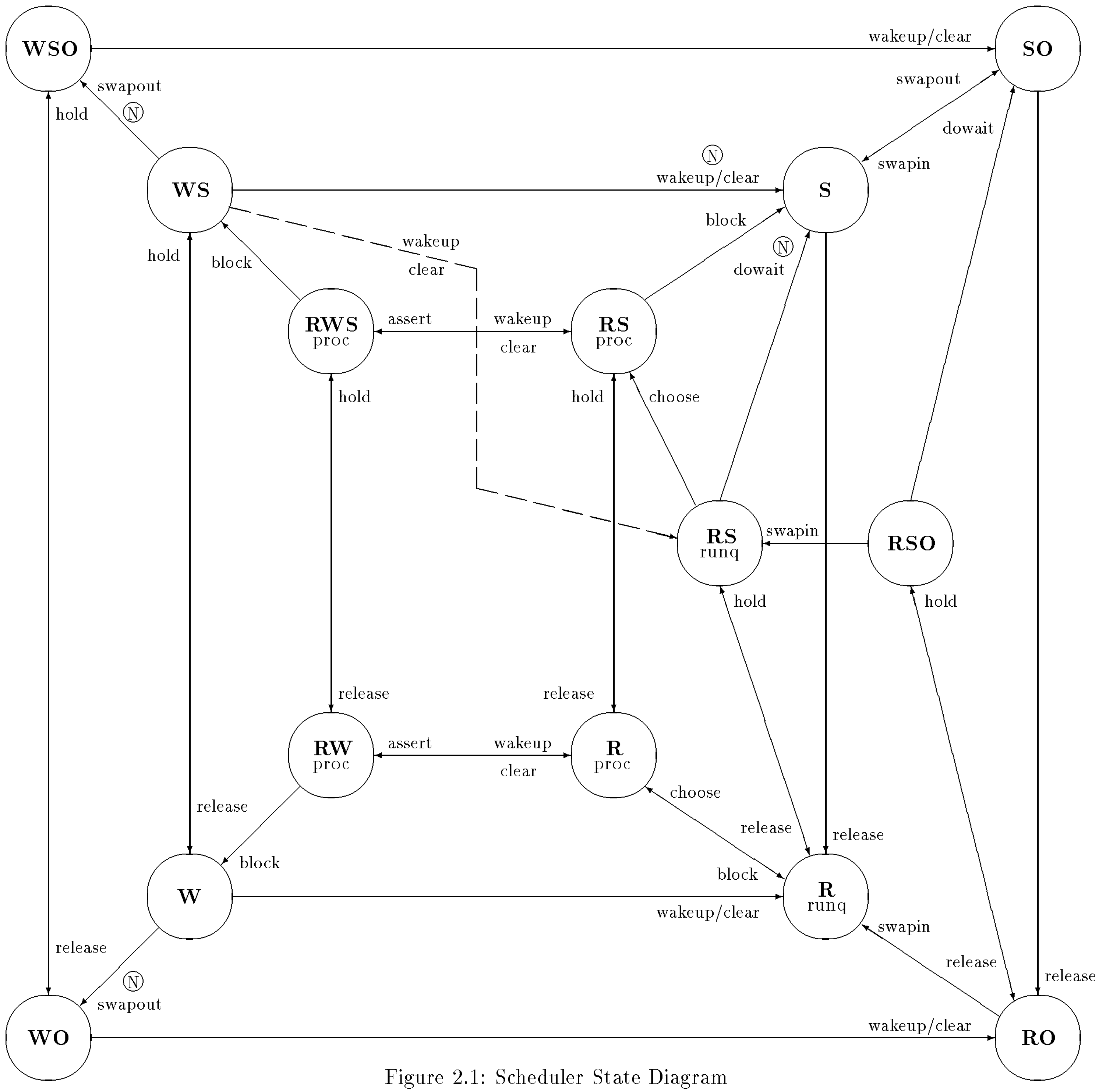

Figure 2.1: Scheduler State Diagram

State	Description
R-proc	Running on a processor.
RS-proc	Running on a processor, suspend pending.
RW-proc	Running on a processor, event wait pending.
RWS-proc	Running on a processor, suspend and event wait pending.
R-runq	Queued to run.
RS-runq	Queued to run, suspend pending.
S	Suspended.
W	Waiting for an event.
WS	Waiting for an event and suspended if interruptible. Waiting with suspend pending if not interruptible.
RO	Swapped out with pending swapin.
RSO	Swapped out with pending suspend and swapin.
SO	Suspended and swapped out.
WO	Waiting and swapped out.
WSO	Waiting, suspended, and swapped out.

Table 2.1: Scheduler States

Label	Routine	Description
assert	**assert_wait**	Declare event to be waited for
wakeup	**thread_wakeup**	Wakeup threads waiting for an event
clear	**clear_wait**	Wakeup a specific thread
hold	**thread_hold**	Hold thread in suspended state
release	**thread_release**	Release one hold on thread
dowait	**thread_dowait**	Synchronize with pending suspend
block	**thread_block**	Context switch, pending suspends and waits take effect
choose	**choose_thread**	Select new thread to run
swapout	**thread_swapout**	Make kernel stack eligible for pageout
swapin	**thread_swapin**	Make kernel stack ineligible for pageout

Table 2.2: Scheduler Transitions

first). There may be multiple overlapping reasons to prevent a thread from executing, so this mechanism employs a suspend count; **thread_hold** ('hold') increments the count, and **thread_release** ('release') decrements it. The 'hold' transitions in the figure occur when the count becomes one, and the release transitions occur when it becomes zero. Other holds and releases do not cause state changes because only the value of the count changes. **thread_block** ('block') causes the suspension to take effect. If one thread is suspending another, **thread_dowait** ('dowait') is used to synchronize the suspension. **thread_dowait** attempts to optimize this synchronization by removing a suspended but runnable thread from its run queue, if possible (RS-runq $\Rightarrow$ S, swapped out threads undergo the corresponding RSO $\Rightarrow$ SO transition); if this cannot be done, it waits for the thread to call **thread_block**. The corresponding primitives exported by the kernel, **thread_suspend** and **thread_resume**, operate on separate thread suspend counts, called user suspend counts, to prevent user actions from interfering with the kernel. The internal suspend count is incremented when the user count is incremented to one; correspondingly, the internal suspend count is decremented when the user suspend count becomes zero. Other user suspends and resumes do not affect the internal suspend count. Threads are initially created in the suspended state and must be resumed in order to execute.

Horizontal transitions in Figure 2.1 involve the event wait mechanism. This is used by threads that need to wait for specific events in the kernel, and is always invoked by the thread itself; any thread can only wait for one event at a time. The wait mechanism uses three primitives; **assert_wait** ('assert'), **thread_block** ('block'), and **thread_wakeup** ('wakeup'). A thread uses **assert_wait** to declare the event it wants to wait on, and **thread_block** to actually wait for it. When the event occurs, it is signalled by **thread_wakeup** ('wakeup). An alternative to **thread_wakeup** is **clear_wait** ('clear') which is based on the identity of the thread that is waiting for the event rather than the event itself; this can be useful if a kernel subsystem (e.g., IPC) internally stores the identities of threads waiting for events that it manages. The use of three primitives (instead of the usual two) allows this event mechanism to be incorporated into other synchronization mechanisms; between the **assert_wait** and the **thread_block**, arbitrary state cleanup may be performed (e.g., unlock locks). If the event should occur while this cleanup is in progress (e.g., a lock is unlocked, allowing another processor to manipulate the corresponding data structure and cause the event), the thread transitions to a state without a W so that **thread_block** has no effect with respect to the event wait mechanism.

The event wait mechanism also supports cleanup of state after the occurrence of an event (e.g., a thread in a device driver may have to restart the device after the completion event occurs). Any thread waiting for an event that requires such cleanup is marked as not interruptible when **assert_wait** is called. In the figure, this disables the transitions marked with a circled N and enables the transition shown with a dashed line. These changes implement two features of threads waiting for non-interruptible events:

1. The thread cannot be interrupted while waiting for the event.

2. The thread will run when the event occurs.

The first effect is implemented by indicating whether each **clear_wait** action is an event completion or an interruption (**thread_wakeup** actions are always event completions), and ignoring interruptions for non-interruptible threads. This means that a non-interruptible event must occur if it is waited for, otherwise the thread that waited for it will be stuck forever. The second effect is implemented by making such threads ineligible for swapout, and by changing how the wait and suspend mechanisms interact for such threads. Suspending a waiting thread (W $\Rightarrow$ WS) normally causes it to enter the S state when its event is signalled; if the thread is not interruptible, it instead enters the RS-runq state (dashed line). The dowait transition from that state is also disabled, forcing the thread to run; its next transition must be to RS-proc. This also changes the meaning of the WS state for non-interruptible threads; for interruptible threads it means 'waiting and suspended', but for non-interruptible threads it means 'waiting with pending suspend' because the thread will run again when the event wait completes. Any running thread that is not in the process of waiting for an event (states R-proc and RS-proc) is always interruptible; **choose_thread** marks all threads entering those states as interruptible. The overall effect of these changes is that non-interruptible threads may only exist in one of six states in three classes; running (RW-proc and RWS-proc), waiting (W and WS), and queued to run (R-runq and RS-runq). Threads in any other state are always interruptible.

Transitions among the three squares of states in Figure 2.1 involve changes in the thread's execution state. Threads move from the inner square to the middle square by calling **thread_block**; this has two effects:

1. Causes pending suspends and event waits to take effect.

2. Reschedules the processor by choosing another thread to run on the processor (**choose_thread**, or 'choose'); this thread moves from a state on the middle square to a state on the inner one.

If there is no pending suspend or wait, **thread_block** decides whether to reschedule the processor based on the priorities of the running and queued threads; if a higher priority thread is queued, a context switch occurs to run it. Threads move from the middle square to the outer square by being swapped out. When physical memory is in short supply, the kernel scans all threads looking for swapout candidates; a thread is a swapout candidate if it has been suspended and/or waiting for at least ten seconds. These threads are swapped out by *unwiring* their kernel stacks, allowing the pageout daemon to page out the underlying memory if needed; the stacks are not explicitly forced out of memory, but are instead entered into the normal page reclaim mechanisms used for all physical memory.

Thread swapping is more complicated than indicated in Figure 2.1. Both swapout and swapin are potentially complex actions that cannot be done atomically (in particular, swapping in a thread may have to wait for physical memory to become available to hold its kernel stack). As a result, a thread is marked as swapped out (in a state containing an O) from the beginning of swapout to the end of swapin. Hence, threads in any swapped out (O) state may have swapout actions in progress, and threads in the RO, RSO, and SO states may also have swapin actions in progress. The thread swapper uses an internal state machine (for each thread) to keep track of the exact state of a thread with respect to the swapper (in particular, an extra state is needed to track threads for which swapin is requested while swapout is in progress to avoid overlapping the two actions).

The scheduling state of a thread is recorded in the state field of the thread data structure. Separate bits in the field represent the R (TH_RUN), W (TH_WAIT), S (TH_SUSP), and O (TH_SWAPPED) components of the state name. The **proc** versus **runq** disambiguation needed for the R and RS states is indicated by the contents of the runq field of the thread data structure; this field contains a pointer to the runq if and only if the thread is on a runq (the -runq states), otherwise it is null (the -proc states). This pointer is needed to determine which runq must be locked when removing the thread from the runq (e.g., the RS-runq $\Rightarrow$ S dowait transition). An additional bit in the state field, TH_IDLE, indicates whether the thread is an idle thread. Idle threads exist only in the R-proc state to absorb processor time that no other thread can use. Earlier versions of Mach queued idle threads on the run queues at the lowest possible priority, but the current version saves their identities in processor data structures from which it is retrieved (by **choose_thread**) when there is nothing else to run on a processor.

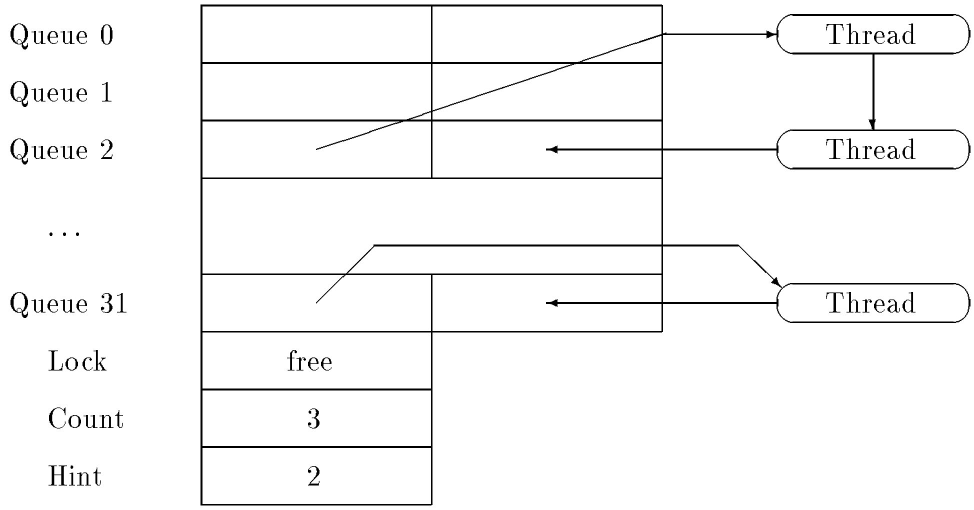

Figure 2.2: Mach Run Queue Structure

2.2.2 Data Structures

The primary data structure used by the Mach scheduler is the *run queue*, a priority queue of runnable threads implemented by an array of doubly-linked queues. Thirty-two queues are used in Mach, so four priorities from the Unix range of 0–127 map to each queue[2]. Lower priorities correspond to higher numbers and vice-versa. A hint is maintained that indicates the probable location of the highest priority thread. The highest priority thread cannot be at a higher priority than the hint, but may be at a lower priority. This allows the search for the highest priority thread to start from the hint instead of the highest priority queue, and can avoid searching a dozen or more queues because the highest possible priority for most user threads is 50 (to match Unix). In addition to the array of queues, the run queue structure also contains a mutual exclusion lock and a count of threads currently enqueued. The count is used to optimize testing for emptiness by replacing a scan of the individual queues with a comparison of the counter to zero. This eliminates the need to hold the run queue lock when the run queue is empty, reducing potential contention for this important lock. The use of a single lock assumes that clock interrupts on the processors of a multiprocessor are not synchronized because significant contention can be expected in the synchronized case, and the use of a shared global run queue may not be appropriate. Figure 2.2 shows an example of a run queue with 3 threads. The queues containing the threads are doubly-linked, but only the forward links are shown for clarity. The hint is 2, indicating that queues 0 and 1 are empty and can be skipped in a search for the highest priority thread.

Each processor consults the appropriate run queues when a new thread is needed for execution. The kernel maintains a *local run queue* for each processor and a shared *global run queue*. The local run queue is used by threads that have been temporarily bound to a specific processor for one of two reasons. Some of the Unix compatibility code in current Mach kernels has not been converted for parallel execution[3] so threads executing in the unparallelized portion of this code are temporarily

[2]More recent Mach kernels use a priority range of 0–31 so that queues and priorities correspond.

[3]Most of this code has been parallelized, see [32] for details.

bound to a single, designated *master processor*. Interrupts that invoke Unix compatibility code are also restricted to this processor. The second use of the local run queues is made by the processor allocation code described in Chapter 6. Mach is self-scheduling in that processors consult the run queues individually when they need a new thread to run instead of having a centralized dispatcher assign threads to processors. The local run queue is examined first to give absolute preference to bound threads over unbound threads. This avoids bottlenecks by maximizing throughput of the unparallelized Unix compatibility code[4], and improves the performance of processor allocation by causing it to preempt other operations. If the local run queue is empty, then the global run queue is examined. In both cases, the highest priority thread found is dequeued and run by the processor. If both run queues are empty, the processor becomes idle.

Special mechanisms are used for idle processors. Most uniprocessor systems borrow the kernel stack of the thread or process that ran most recently to execute the idle loop. On a multiprocessor this can be disastrous if the thread resumes execution on another processor because the two processors will corrupt the thread's kernel stack. Mach avoids this stack corruption by using a dedicated *idle thread* for each processor. Idle processors are dispatched by a mechanism that bypasses the run queues. The kernel maintains an internal *idle queue* of processors. A thread that becomes runnable dequeues a processor from that queue and enters its identity in that processor's data structure. The idle thread on that processor picks up the new thread and context switches directly to it, gaining performance by bypassing the run queues. This mechanism is one example of a general technique called Handoff Scheduling that context switches directly to a new thread without searching for it on a run queue.

A recent paper by Anderson, et al[1] explores data structure alternatives for a user-level thread package; their results provide some guidance for the design of a multiprocessor scheduler. They considered three alternatives for the run queue and idle processor management data structures:

1. A global run queue.
2. A global run queue plus a queue of idle processors.
3. A run queue per processor:
   - Newly runnable threads are assigned to random run queues.
   - Processors search queues in cyclic order starting with their own.

The evaluation used a program that created and destroyed one million fine grained threads to obtain speedup and latency results. The speedup results are not directly applicable to Mach because thread creation should not be a major performance bottleneck; applications using medium to fine grain parallelism can be expected to cache and reuse operating system threads or processes (as their benchmark does) to avoid the expense of creating and destroying them. The latency results are applicable, as they should be indicative of the scheduling overhead of blocking and resuming threads; these are important operations in any operating system, and particularly for one like Mach that supports message passing as a primary means of interaction among threads.

The results from this study suggest that the data structures used in Mach are the best choice among the alternatives. Compared to the base case of a single global run queue, adding an idle queue significantly improved performance when the number of runnable threads was less than the number of processors, but imposed a slight penalty when there are more threads than processors. In the former case, threads are a scarce resource, and it is more efficient to have them looking for

[4] See [55] for an extensive study of this area.

processors than vice versa. In the latter case, there is a slight overhead for checking the idle queue to discover that it is empty. The run queue per processor alternative made no difference for small or large numbers of threads, but had a significant negative impact if the number of threads was close to the number of processors because each processor would search more run queues in looking for a thread. Recent work on the performance benefits of limiting the parallel execution demanded by applications to the available hardware parallelism suggests that a multiprocessor operating system like Mach can expect to spend much of its time in a condition where the number of threads and the number of processors roughly correspond [51, 54]. This makes the performance of the run queue per processor solution unacceptable for a multiprocessor operating system. This and the performance benefits of the idle processor queue (when there are fewer threads than processors) suggest that the global run queue plus idle queue alternative used by Mach is the best choice.

2.2.3 Priority Calculations

Threads are chosen for execution based on their priorities, and the scheduler adjusts these priorities to balance processor usage among competing threads. Thread priorities consist of a base priority plus an offset derived from recent processor usage. The base priority is set to allow internal kernel threads to run at higher priorities than user threads because the internal threads perform critical kernel functions such as pageout. The offset derived from usage is weighted by the load factor to preserve system responsiveness under load. If an adequate hardware source of timestamps exists, such as a 32-bit microsecond counter, the scheduler can be configured to base usage calculations on timestamps from this counter. This eliminates the inaccuracies and distortions caused by statistical usage calculations (see [55] for details).

Mach uses the overload factor technique for processor usage aging. The aging overhead is distributed by making each thread responsible for aging of its own processor usage. Clock interrupts and events that unblock a thread cause it to check its local copy of a counter against a global value that is incremented once a second. If these values differ, the thread ages its usage by a factor of 5/8 for each unit of difference. This results in each thread's accumulated usage being multiplied by 5/8 once a second. 5/8 was chosen for efficiency (multiplication by it can be implemented by two shifts and an add), and to produce behavior similar to other timesharing systems[5]. Mach uses an overload factor of the number of runnable threads divided by the number of processors with a minimum of 1. This factor is calculated as an exponential average to smooth the impact of abrupt load changes.

This scheduling algorithm requires that all threads check their copy of the counter on a regular basis. The clock interrupt handler performs this check for running threads, and threads that are not running defer this check until they become runnable. Runnable threads with low priorities may spend long periods of time on a run queue while higher priority threads monopolize the processor. Such threads would run if they could check their copy of the counter and age their usage, but their low priority positions on the run queue prevent them from running to perform this check. To avoid the resulting starvation, an internal kernel thread scans the run queues once every two seconds to perform the aging and resulting priority elevation for any threads in this situation. The two second period was chosen to produce acceptable behavior based on experience with versions of the system that exhibited this starvation. Similar starvation has been observed in the ESCHED scheduler due to processes failing to cause a sufficient rate of priority-elevating events. A similar solution of using a periodic scan of the run queue to boost priorities of such processes is described by Essick[15].

[5]The number used by 4.3BSD Unix depends on load. For the range from 0.5 to 1.0, it varies from 1/2 to 2/3[34].

Expansion	Approximation	Result
0.1000	$2^{-2} + 2^{-2}$	Exact
0.1001	$2^{-1} + 2^{-4}$	Exact
0.1010	$2^{-1} + 2^{-3}$	Exact
0.1011	$2^{-1} + 2^{-3}$	Error of 9.1%
0.1100	$2^{-1} + 2^{-2}$	Exact
0.1101	$2^{-1} + 2^{-2}$	Error of 7.7%
0.1110	$2^{0} - 2^{-3}$	Exact
0.1111	$2^{0} - 2^{-4}$	Exact

Table 2.3: Binary Fractional Approximation

A general technique involving two shifts and an add or subtract is used to approximate all fractional multiplications in the scheduler. Multiplication by 5/8 is implemented exactly (shift right 1, shift right 3, add). The worst possible error of this approximation is 9.1% and the expected (average) error is less than 3% with the overall error distribution being unbiased (i.e., symmetric around 0). This accuracy is more than sufficient for scheduling purposes, especially since the most common operation, multiplication by 5/8, is implemented exactly. The remaining operations, multiplications by powers of 5/8, are approximated. The accuracy of this approximation is derived from its ability to match the first four digits of the binary expansion of a fraction in 75% of the cases. This is shown in Table 2.3 for the expansions between 0.5 and 1. For other fractions, the first three or four digits are matched in a similar fashion. The 3% error bound is obtained by extending the expansions that are matched exactly to one or two additional places and then performing piecewise integration of the resulting error function; the result is an upper bound on the average case error and is less than 3%.

2.2.4 Priorities and Usage

An important characteristic of a timesharing scheduler is how priorities affect the usage of processors, in particular, how processor time is divided among threads at different priorities. This problem is addressed here under two simplifying assumptions:

1. The threads involved are entirely compute-bound.
2. A continuous approximation is used to obtain steady-state results.

These results are indicative of how threads share the processors, but do not provide any insight into transient behavior of the scheduler or how threads that block on a regular basis will behave. These results also do not address the question of whether the scheduler will converge to the steady state solution (in practice it seems to always do so), but provide some information on the convergence rate.

A fundamental constant that factors into all of the problems considered here is the priority drop caused by the accumulation of one second of usage, P. For a priority range of 0–31, usage times computed in microseconds are shifted right 18 bits before being added to the corresponding priorities. Thus:

$$P_{32} = \frac{10^6}{2^{18}} = 3.8147...$$

For a priority range of 0–127, the shift is 16 bits instead of 18 bits, so:

$$P_{128} = \frac{10^6}{2^{16}} = 15.2587...$$

Due to the approximations in the scheduler, the values of P are not significant beyond a few decimal places. This quantity is carried through the analysis as the symbol P to show the dependence of these results on it. The use of accurate timestamp sources that are not denominated in microseconds requires changes to P as described in the next chapter.

We now consider how processors are shared by compute bound threads at the same priority to show that the steady-state scheduler behavior is invariant with respect to the number of threads and number of processors. For a single thread at priority p, accumulated usage will cause its steady-state priority to be lower than p (i.e., a larger number) by a quantity we denote by d. Since the thread's usage is multiplied by 5/8 once a second, these quantities are related by:

$$\frac{5}{8}(d + P) = d$$

$$d = \frac{5}{3}P$$

If more than one thread is involved, then each thread only gets a fraction of the processor time. We use the notation f_i for the fraction of time obtained by thread i. If there are n threads involved, then the above equation becomes:

$$\frac{5}{8}(d_i + Pf_i l) = d_i$$

where the l is the overload factor, which is n in this case. If all of the base priorities, p_i, are equal, then the f_i must all be $1/n$ in steady state, so this reduces to the previous case:

$$d_i = \frac{5}{3}P, \ i = 1, ..., n$$

If m processors are involved, there are two possible cases. For $m \geq n$, each thread has its own processor and this reduces to the single thread case for each thread. For $m < n$, the overload factor, l, is n/m, so we have:

$$\frac{5}{8}(d_i + Pf_i\frac{n}{m}) = d_i$$

For equal priorities, the f_i must be m/n in steady state, so this reduces to the previous multiple thread result. Hence the depressions, d_i, are always $\frac{5}{3}P$, independent of the number of threads and processors involved. For $P = 3.8$, this means that a compute-bound thread in an environment consisting only of compute-bound threads can expect to see its priority depressed by a total of 6 (actually 6.3 before rounding).

The general problem involves sharing of processors by threads at different priorities. If n threads are at priorities p_i, we are interested in solving for the corresponding usage fractions f_i. Without loss of generality, assume that thread n has the lowest priority, and let $e_i = p_i - p_n$ represent the elevations in priority for the other threads. On a multiprocessor we still have the steady-state usage equation for each thread:

$$\frac{5}{8}(d_i + Pf_i\frac{n}{m}) = d_i$$

Solving for d_i, we obtain:

$$d_i = \frac{5}{3}Pf_i\frac{n}{m}$$

The elevations and depressions are related because a thread must drop in priority by its elevation to be treated the same as thread n, hence $d_i = d_n + e_i$. Substituting this and the previous identity for d_n, we can solve for f_i in terms of f_n:

$$\frac{5}{3} P f_n \frac{n}{m} + e_i = \frac{5}{3} P f_i \frac{n}{m}$$

$$f_n + \frac{3}{5} \frac{e_i m}{nP} = f_i$$

This formula indicates how the fractions of processor time for two threads whose priority differs by e_i will differ. To solve for the actual fractions, we note that they must sum to the number of processors, and obtain:

$$\sum_{i=1}^{n} \left(f_n + \frac{3}{5} \frac{e_i}{nP} \right) = m$$

$$f_n = \frac{m}{n} - \frac{3m}{5n^2 P} \sum_{i=1}^{n} e_i$$

This solution has made the implicit assumption that all threads are executing. To check this we impose the boundary condition that the lowest priority thread, n, must obtain some processor time and derive a necessary and sufficient condition on the priority elevations:

$$\frac{m}{n} - \frac{3m}{5n^2 P} \sum_{i=1}^{n} e_i > 0$$

$$\frac{5}{3} nP > \sum_{i=1}^{n} e_i$$

If this condition is violated, one or more of the lowest priority threads do not execute. Solutions to specific cases can be found by substituting actual values. For two threads ($n = 2$) differing in priority by 1 ($e_1 = 1$, $e_2 = 0$) on a uniprocessor ($m = 1$) with the Mach scheduler ($P = 3.8$), the resulting f_i differ by about 8%, so the higher priority thread can expect an additional 8% of the processor compared to the lower priority thread.

2.2.5 Priority Interface

Mach exports an interface that allows the priorities of threads to be controlled by applications. The priority exported by this interface is the base priority for the calculations discussed previously. Each thread has both a priority and a maximum priority. The priority must never be greater than the maximum priority, and the maximum priority can only be decreased by unprivileged applications. Tasks also have priorities, which are used to initialize the priorities of newly created threads, but the interface routine that changes a task priority includes the option of changing the priority of all threads currently in that task. A newly created thread inherits its initial priority from its task, and its initial maximum priority is set by the system. Tasks in turn inherit their priority from their parents at creation time. The default maximum priority for threads and the priority for the first task (which all other tasks will inherit in the absence of changes) are currently set to 50 (the default priority value used by Unix). This interface is described in Section 6.3 and Appendix B, because it interacts with and depends upon the kernel interface for processor allocation. This allows a user who has allocated dedicated processors to exercise complete control over the scheduling on those processors for the duration of the allocation.

2.2.6 Managing Context Switches

The scheduler utilizes time quanta to minimize the number of context switches it causes. When a processor begins to run a thread, that thread is assigned a time quantum. Periodic clock interrupts decrement this quantum, and a rescheduling check occurs when the quantum expires if the thread does not stop or block before then. This rescheduling check causes a context switch if there is another thread to run of equal or higher priority, otherwise the original thread continues running with a new quantum. The context switch mechanism uses an AST (Asynchronous System Trap) to avoid context switches in interrupt service routines. An AST is a general mechanism that allows an interrupt service routine to force the current thread to trap into the kernel from user mode. AST's may be directly supported in hardware, implemented entirely in software, or a combination of the two. Priority reductions due to processor usage accumulated during a thread's first quantum do not cause context switches. This feature reduces the number of context switches for load and usage balancing by extending the time period between context switches.

An additional reduction in the number of context switches is achieved on multiprocessors by lengthening the quanta when the number of runnable threads is less than twice the number of processors. With a minimum quantum of 100 milliseconds, each processor on a 10 processor machine with 11 runnable threads will context switch 10 times a second. For threads, this results in 10ms typical duration on the run queue, which is much smaller than the corresponding 100ms value for a uniprocessor in the corresponding situation (two runnable threads). Similar reductions in dwell times on the run queue occur in any situation where the number of runnable threads is less than twice the number of processors; for more threads, the maximum context switch rate on all processors yields dwell times of the minimum quantum or more. Mach reduces the number of context switches in such cases by extending quanta so that the expected time on the run queue is the minimum quantum. In the 10 processors and 11 threads example, a quantum of 1 second would be used to achieve an expected time on the run queue of 100ms; this cuts the number of load balancing context switches by a factor of 10. Care must be taken in assigning these longer quanta to avoid bunching of the processors. In this example, it would be possible to assign one second quanta to all of the processors within one clock period of each other, resulting in an unacceptable wait of nearly one second on the run queue for at least one thread. The scheduler assigns fractions of the extended quanta (instead of the entire quanta), as needed to prevent this behavior.

Context switches are also used to implement preemption of lower priority threads when higher priority threads become runnable. Preemption occurs immediately on a uniprocessor where an AST is requested as part of making the higher priority thread runnable, but it may take up to a clock interrupt period on a multiprocessor if another processor must notice the presence of a higher priority thread on a run queue. Mach does not use interprocessor interrupts for preemption because they haven't been necessary to achieve acceptable timesharing behavior. A consequence of this is that the scheduler does not maintain data structures that would permit the efficient identification of the lowest priority processor when preemption is needed. The use of interprocessor interrupts for preemption is expected to be added in the future as Mach evolves to include support for some real-time applications.

Chapter 3

Accurate Usage Timing

Accurate usage measurements are important to both the management and use of uniprocessors and multiprocessors, but current statistical measurement techniques produce inaccurate and distorted results. The Mach timesharing scheduler uses these measurements to adjust priorities for load and usage balancing; inaccuracies and distortions in these measurements can result in bad balancing decisions. Timing information is also of vital importance for measuring the performance of application programs including measures such as efficiency and speedup. Current statistical timing techniques often suffer from accuracy problems for short intervals and exhibit consistent distortions in their results. These problems are caused by the use of clock interrupts to collect the samples on which the measurements are based. Improving these measurements requires a different technique.

This chapter describes a new timestamp-based technique for accurate measurement of execution time that has been developed for Mach. The resulting measurements are far more accurate than those derived from statistical timing and are free of the associated distortions. On the other hand, there is an additional cost in overhead, but it can be reduced to an acceptable level by careful design and implementation. An important component of this design is the use of algorithms for concurrent reading and writing without mutual exclusion developed by Lamport [30, 31]. Measurements of the Multimax implementation quantify the overhead increase caused by these measurement techniques and show that it is small.

3.1 Statistical Usage Timing

Statistical timing is based on sampling the use of each processor and assuming that the sample is representative. This is the most common technique for measuring processor usage, and is usually implemented by a clock that generates interrupts at a known rate (100Hz is often used). Code in the interrupt handler determines what was interrupted (e.g., which thread, user or kernel mode) and charges it with one tick of usage (the period since the last interrupt, 10ms for a 100Hz clock). In the statistical sense, the use of the processor is sampled by each clock interrupt; over reasonable periods of time the samples should provide a good approximation to actual usage in the absence of sampling anomalies. This is adequate for scheduling purposes; improvements could be obtained from more accurate information, but a large number of successful timesharing systems are based on this technique.

There are a number of drawbacks to statistical timing. It is designed for measuring long term averages, so its precision for short term measurements is limited by the clock interrupt rate. Many systems have been reducing this clock interrupt rate to reduce overhead. One example is the Mach implementation on the Encore Multimax which uses a clock interrupt rate of 10Hz instead of the usual Unix rate of 100Hz. Such reductions cause a loss of precision in timing measurements and lengthen the period over which they must be averaged to attain a given degree of accuracy. These measurements are of no use in measuring times that are on the order of a few seconds or less, but such measurements are of great importance in a multiprocessor because it may be difficult or impossible to reliably reproduce short-term synchronization behavior that has a major impact on performance. Statistical timing also suffers from a distortion problem, in that the overhead of the clock interrupt routine and any activities synchronized to it (e.g., network protocol timeout processing) are invisible. Such activities usually complete before the next clock interrupt and are therefore never interrupted and sampled. As a result, the time consumed by such activities is hidden in the time charged to other uses of the processor, resulting in a consistent overestimation of usage times by several percent. This effect can be aggravated by usage patterns that happen to synchronize with the clock interrupt frequency; in the worst cases, the resulting usage measurements may bear no resemblance to the actual usage pattern [55]. For an extreme example reported by Wendorf [55], the usage times measured for a specially constructed pair of programs differed from the actual times by factors of more than 20.

3.2 Timestamp-Based Usage Timing

Timestamp-based timing tracks what each processor executes and accumulates time measurements accordingly. This increases both the the precision and accuracy of the measurements, in part by avoiding the distortions of statistical timing. Timestamp-based timing is based on the abstraction of a *timer*, which is similar to a stopwatch. A timer can be started or stopped, but can be reset after its initialization. To account for all processor time, the timing system ensures that every processor is always accumulating time on exactly one timer. Each thread in the system has two timers, one for user mode execution and one for system mode execution. In addition, each processor in the system has at least one timer for execution that does not occur in a thread context (interrupt routines); more than one of these *kernel timers* can be used to distinguish among these activities (e.g., different interrupts). Operating system code is responsible for switching among timers to make sure that the correct timer is accumulating time for each processor. There are five locations in the operating system where such switches must be made:

1. In response to a trap, fault, or system call from user mode: switch to the system timer for the thread.

2. As part of returning to user mode from a trap, fault, or system call: switch to the user timer for the thread.

3. In response to an interrupt: switch to the kernel timer for the interrupt.

4. As part of returning from an interrupt: switch back to the timer that was running before the interrupt.

5. As part of a context switch: switch to the system timer for the new thread.

These switches ensure that user timers accumulate execution time in user mode, system timers accumulate execution time in system mode, and kernel timers accumulate kernel overhead time. Idle time is accumulated by system timers for the idle threads.

There are several problems that have limited the use of timestamp-based timing. The most serious problem is that many hardware implementations do not provide a source of timestamps.

This is a simple omission that should be corrected in the hardware, as it is relatively cheap and easy to provide a counter that runs at some known clock rate. This counter should be provided in a page of memory that can be mapped into applications to allow them to obtain timestamps efficiently[1]. Even if the appropriate hardware support is present, the increased accuracy of the resulting measurements makes them more complicated to manage. Statistical usage timing can use a single word of memory to hold the accumulated tick count, and can rely on memory subsystem hardware to synchronize accesses from multiple processors. The additional precision present in timestamp-based measurements requires that more than one word of memory be used to hold accumulated time, and therefore requires software protocols to manage concurrent accesses to this memory. The problem is compounded by the fact that code to update timestamp-based times occurs in critical paths of the kernel such as trap, interrupt, and system call processing. This makes performance a paramount concern, and the use of mutual exclusion locking (e.g., test and set) undesirable.

3.3 The Mach Timing Facility

The Mach operating system has been modified to incorporate a timestamp-based usage timing facility for the reasons noted above. The software problems that have limited past usage of this technique have been overcome by careful design and implementation. The resulting facility uses no multiprocessor interlock instructions to control access to its stored times, and is optimized to yield short code paths for the common critical operations. Appropriate hardware support is still a requirement. The initial implementation of this facility was done for the Encore Multimax, and used its 32-bit free running microsecond counter as a timestamp source.

3.3.1 Design

The design of Mach's timestamp-based timing facility involves one data structure with four operations, the *timer*. This can be thought of as a stopwatch; at any time it is either running and accumulating time or stopped. When restarted, it continues from the point at which it was stopped. Each thread has a timer for user and system time, and in addition, the kernel has one or more timers for interrupt-related activities. At any time there is exactly one timer running for each processor in the system; a user timer if the processor is executing a thread in user mode, a system timer if the processor is executing a thread in kernel mode, or a kernel timer if the processor is executing an interrupt handler. Idle processors accumulate time on the system timer for their idle thread. A timer consists of two words of memory to hold the accumulated time, a check field used to avoid access conflicts, and a timestamp field to hold the timestamp that started the timer running. Two words of memory are needed to hold the accumulated time in a timer because one word overflows too quickly. For a common case of microsecond timestamps and 32-bit words, the overflow period for one word is about 70 minutes.

There are four basic operations on timers:

1. **Start.** Start a timer running.

2. **Update.** Stop a timer and update the time it contains.

3. **Normalize.** Normalize the accumulated time values.

[1] Mach's *device pager* can be use to implement this mapping.

4. **Read**. Read the value in a timer.

The **Start** operation is used during the boot sequence to start the initial timer for each processor. Thereafter, each trap, trap return, interrupt, interrupt return, and context switch performs an **Update** operation on the current timer and a **Start** operation on a new one using the same timestamp. This combination of **Update** and **Start** with the same timestamp is called **Switch**. Traps and trap returns that proceed from system mode to system mode avoid these operations because they do not change the running timer. Interrupts must be disabled when switching between timers to make the **Switch** operations atomic so that the invariant of one timer running at any time on each processor can be maintained. The **Normalize** operation moves time within the timer between fields denominated in different time units (e.g., seconds, microseconds) to avoid overflow. It is invoked as needed by **Switch** operations. **Read** operations are used in user-invoked system calls to read the value of a timer. The scheduler uses an optimized version of **Read**, called **Delta** to maintain the usage information that drives the scheduling algorithm.

Times are stored in a non-normalized format using units appropriate to the timestamp source to make the **Switch** operation as fast as possible. The non-normalized format permits the low order word of accumulated time to hold more than one of the units of the high order word. For example, if the units for the low and high order words are microseconds and seconds, respectively, then the microsecond word may hold more than one million microseconds. The units for the low order word are chosen to match the timestamp source, so an **Update** involves a subtraction to calculate the elapsed time and an addition to add it to the low order word accumulated time field; no conversion or normalization operations are required. Units are converted if needed (e.g., if the timestamps come from an 8MHz clock, division by 8 is necessary) by **Read** operations because they are less frequent and do not occur in critical paths. The special **Delta** operation used by the scheduler does not perform unit conversion; instead the scheduler keeps its internal statistics in the same units as the hardware timestamps. Conversion from these machine dependent units occurs in the scheduler's priority update calculations via shift and add operations. Up to two shifts and an add can be configured to implement any desired conversion factor. Normalization overhead is reduced by delaying normalization until the low order word of the accumulated time exceeds 2^{31} (for 32-bit words); this is about 35 minutes of accumulated time for microsecond timestamps. This overflow is checked by each **Update** operation, except that an **Update** caused by an interrupt defers this overflow check until the corresponding interrupt return to provide an improvement to interrupt response time.

Conflicting write accesses to timers are eliminated by restricting write operations (**Switch** and **Normalize**) to a single writer for each timer. For a usage timer associated with a thread, the writer is that thread. For kernel timers, this is achieved by associating each timer with a specific processor. As a result, kernel timers are grouped in arrays with one timer for each processor on a multiprocessor. Currently, a single set of kernel overhead timers is used, but different sets could be associated with different interrupts. This would yield information on how much time is spent servicing each interrupt source and how that time is distributed among the processors.

Read/write conflicts are simplified by defining the **Read** operation so that most conflicts are resolved by hardware. The **Read** operation is defined to only return time from the accumulated time fields; time since the last **Start** operation for a running timer is ignored. The clock interrupt rate limits the resulting inaccuracy; for a 100Hz clock rate, no more than 10 milliseconds will elapse before the next event that updates a running timer (context switch, trap, trap return, interrupt, interrupt return). More accurate values can be obtained by stopping the timer (e.g., suspend the corresponding thread) or causing an event to occur (e.g., interprocessor interrupt) if this is important. This definition of the **Read** operation makes the timestamp field of the timer private to the **Start** operation, removing it from participation in read/write conflicts. The vast majority of read/write conflicts are with an **Update** operation on the write side, and hence only involve the low order accumulated time field. The memory hardware resolves these conflicts by ordering accesses to this field. In both cases, the **Read** operation returns valid data because the order of the accesses determines whether the **Read** appears to have occurred before or after the **Update**.

Conflicts with the **Normalize** operation are resolved by a backout protocol. Because the **Normalize** operation writes both accumulated time fields, it is possible for a naive **Read** operation to obtain one pre-normalization value and one post-normalization value. This is avoided by using the check field to implement a backout protocol that combines features of Lamport's Two-Digit monotonic clock algorithm[31] and his solution to the single writer case of the readers/writers problem[30]. The check field holds a copy of the high order accumulated time field of the timer. **Normalize** operations write the normalized accumulated values in the order check-low-high. **Read** operations read the fields in the reverse order, high-low-check, and test to see if the high and check fields match. No **Normalize** operation could have been in progress if they match[2], so the data read from the timer is valid. If the high and check values don't match, the values read from the timer have been corrupted by the **Normalize** (the low value corresponds to one of the two high values, but it's impossible to tell which one), so the **Read** operation is retried. This retry always succeeds because **Normalize** operations are infrequent.

Figure 3.1 shows pseudo-code for the major timer operations. For simplicity, the timestamp units are assumed to be microseconds (the base unit of the **Read** operation); if this is not the case, the **Read** code would contain additional conversion code to change units. This example keeps the high_bits field of the timer in seconds for convenience; **Normalize** operations are infrequent enough that choosing the unit for the convenience of **Read** makes sense. The actual interface used within the kernel to access and manipulate the timers is documented in Appendix E.

The scheduler's **Delta** operation (optimized **Read**) uses a copy or *snapshot* of the timer to return elapsed time. The operation compares this snapshot to the current timer value, returns the elapsed time, and updates the snapshot. An efficient implementation is obtained by observing that the high order field of the snapshot will usually match the timer; only the low order fields of the timer and snapshot are involved in most instances. The implementation reads the low field, and then compares the high field of the snapshot with the check field of the timer. This is the high-low-check order of accesses used by a **Read** operation except that the high value comes from the last snapshot instead of the current operation. If the high values match, then the difference in low values is the elapsed time and the new snapshot is obtained from the old by updating the low field. If the high values don't match, a **Read** operation is performed to obtain a new snapshot from which the elapsed time is then calculated. This **Read** operation is overkill in the current system because **Delta** operations are only executed by the thread on its own timers or by some other thread when the target is known not to be running; in both cases, **Normalize** operations are implicitly excluded. The performance impact of these **Read** operations is minimal because they only occur after **Normalize** operations, which are rare. The resulting implementation is extensible to schedulers that do need to read elapsed times from running threads.

3.3.2 Relationship to Lamport Clocks

The algorithms used in Mach's timing facility are based on algorithms developed by Lamport, but there are several differences and extensions. Lamport's two-digit monotonic clock algorithm[31]

---

[2]This assumes that performance of write operations cannot be reordered by hardware. For hardware that violates this assumption (e.g., PLUS[6]), fence operations must be inserted between the writes performed by the **Normalize**.

```
struct timer {
        unsigned low_bits;
        unsigned high_bits;
        unsigned high_bits_check;
        unsigned tstamp;
} *timer_t;

#define TIMER_HIGH_UNIT 1000000

Start(timer, timestamp)
timer_t timer;
unsigned timestamp;
{       timer->tstamp = timestamp; }

Update(timer,timestamp)
timer_t timer;
unsigned timestamp;
{       timer->low_bits += timestamp - timer->tstamp; }

Normalize(timer)
timer_t timer;
{
        unsigned        high_increment;

        high_increment = timer->low_bits/TIMER_HIGH_UNIT;
        timer->high_bits_check += high_increment;
        timer->low_bits %= TIMER_HIGH_UNIT;
        timer->high_bits += high_increment;
}

Read(timer, seconds, microseconds)
timer_t timer;
unsigned *seconds;
unsigned *microseconds;
{
        unsigned save_high, save_low;

        do {
                save_high = timer->high_bits;
                save_low = timer->low_bits
        } while (save_high != timer->high_bits_check);

        *microseconds = save_low%TIMER_HIGH_UNIT;
        *seconds = save_high + save_low/TIMER_HIGH_UNIT;
}
```

Figure 3.1: Timer Operations

maintains a clock in normalized format, and uses this to allow the reader to infer a valid clock value if a read and a write should overlap. Mach deliberately uses a non-normalized format to favor writes over reads by limiting the effect of write operations. The cost of not using a normalized format is that it is no longer possible to infer a valid clock value if a read operation detects an overlapping write (which must be a **Normalize**). Instead the read must be retried, as per Lamport's general readers/writers solution[30]. The rarity of normalization operations in this case guarantees that the retry will succeed. In addition, the application of this technology to usage timing is new because the motivating application for Lamport's work was the difficulties encountered in sharing multiple word time of day clocks between applications and the operating system. The use of a lock that can be held by applications or the kernel is clearly unacceptable because it puts the kernel at the mercy of a malicious application. Work is in progress to use Lamport's monotonic clock algorithm to implement this feature (time of day clock in shared memory) for Mach.

3.3.3 Implementation

There are several options for timestamp sources. The Multimax implementation uses a shared free running 32-bit microsecond counter as the timestamp source; reading this counter bypasses the processor caches and takes 2.5 microseconds (slower than a memory access that misses in cache). All accesses to this counter are serialized by the bus protocol, so it is a potential source of multiprocessor contention. An alternative hardware design is to provide a single timestamp source for each processor in a multiprocessor. Since timestamps are only compared to others from the same processor, it is not necessary to synchronize these sources. In other words, implementing a distributed timestamp source for the Mach timing facility does not require synchronized clocks. Counters that roll over at values other than 2^{32} can be used; the rollover period must be long enough that it can be detected reliably. The detection mechanism notices that the difference between two timestamps is negative, infers that exactly one rollover has occurred, and compensates by adding the rollover period. A useful practical criterion for reliable detection is that each processor's counter sampling rate (for timestamps) should be at least twice the rollover rate. A periodic countdown counter that generates an interrupt on rollover is not sufficient without an additional rollover detection mechanism (e.g., a bit set by hardware when the rollover occurs). In the absence of such a mechanism, delays in accepting clock interrupts may cause the timing facility to miss rollovers. This happens when a rollover interrupt is delayed to the extent that its interrupt entry timestamp is greater than the interrupt exit timestamp of the previous interrupt's service routine. Generating an additional interrupt at the midpoint of the counter's rollover period is another way to avoid this problem.

The Multimax implementation uses inline assembly language for the timing operations in critical paths. This uses approximately one third fewer instructions than a previous implementation that used C routines to manipulate the timers. The major savings in instruction count for the assembly language version come from avoiding the instructions that implement procedure call (three on entry, three on exit). An additional savings is realized in the trap cases by optimizing the protocol used to disable interrupts. The Multimax enables and disables interrupts via an off-chip ICU, but a chip bug requires that the processor internally disable interrupts before manipulating the ICU and re-enable them afterwards. This results in three instructions for each enable and disable, or a total of 6 instructions per timing operation. The timing code for traps reduces this to two instructions per operation by disabling interrupts on the processor; this technique has to be used with great care because non-maskable interrupts are also routed through the ICU, so disabling interrupts on the processor also disables non-maskable interrupts.

Operation	Instructions	Min	Max	Average	Typical
Trap Entry	15	13.5	27.5	14.1	13.5
Trap Exit	15	12.5	26.5	13.2	12.5
Interrupt Entry	12	10.5	21.5	10.7	10.5
Interrupt Exit	15	13.5	28.5	14.0	13.5
Context Switch	13	9.5	24.5	11.9	10.5

Table 3.1: Multimax Timer Switch Costs (Instructions and Microseconds)

3.3.4 Performance

Timestamp-based timing can achieve orders of magnitude improvements in timing accuracy. On a Multimax with a 10Hz clock interrupt rate, statistical timing is of little use for periods of less than several seconds; at one second, the intrinsic measurement error is 10%, and this drops to 2% at five seconds. In contrast, Mach's timing facility can time periods of less than one hundred microseconds on the same hardware with excellent accuracy and repeatability. This represents an improvement more than four orders of magnitude (10^4) in the precision of the timing facility. For a more common 100Hz clock rate, the corresponding improvement with a microsecond timer is in excess of three orders of magnitude. Additional improvements in accuracy are obtained by correctly excluding kernel overhead from user timing measurements. These improvements in precision and accuracy have delivered valuable new timing capabilities to users at CMU and elsewhere.

Timing overhead measurements were taken on the Multimax implementation for the **Switch** operations in the critical paths of the kernel. Because these measurements rely on the shared counter to compute elapsed times, every critical path accesses the counter three times during these tests, causing the results to overstate the effects of access contention for the counter. Each operation was measured 2048 times on a Multimax with ns32332 processors (about 2 MIPS each) to obtain a representative time distribution. These results (in microseconds) and the assembly language instruction count for each operation are shown in Table 3.1. The Typical times are both the median and mode of the corresponding distributions. All of the operations in the table include an overflow check, except that the overflow check for Interrupt Entry is postponed until the corresponding Interrupt Exit.

The results of these experiments show that low timing overheads can be achieved in practice. The typical operation times of 10 to 15 microseconds from Table 3.1 are cheap in comparison to other costs in the system. For example, the fastest context switch path from user mode to user mode takes 320 microseconds on this machine; the timing code accounts for about 36 microseconds, or 11.25%. This is the worst-case; for a reasonable system operation time of about 1 millisecond, the corresponding overhead is about 3.5%, and the percentage decreases as user-mode computation time is considered (e.g., for a typical user computation time of 3 milliseconds between system operations, the overhead is less than 1%). These small overheads are more than justified by the orders of magnitude increase in accuracy and the absence of distortion in the results produced by Mach's timing facility[3].

These overhead measurements generalize to other architectures, but care must be exercised in making the generalizations. The time per instruction is not uniform because it takes 2.5 mi-

[3]See Appendix A of [55] for details on the potential inaccuracies and distortion of statistical timing.

croseconds to obtain a timestamp. For example, the 10.5 microsecond typical time for the **Switch** operation in the Context Switch path consists of 8 microseconds to execute 12 instructions plus 2.5 microseconds to obtain the required timestamp. The Trap Entry and Trap Exit code sequences assume that the trap is known to be from user mode; two extra instructions are needed to determine this when it is not known. Similarly the Interrupt Entry and Exit sequences assume that all interrupts are blocked; two extra instructions would be needed if this is not the case. The Context Switch code uses an extra instruction that references memory due to a compiler convention that imposes a limit of three scratch registers.

3.4 Conclusion

This chapter has shown that the goals of low overhead and accuracy for a processor usage timing facility are not mutually exclusive when a minimal level of hardware support is present. By comparison to statistical timing techniques, the Mach timing facility achieves orders of magnitude improvements in accuracy and all but eliminates distortion with an overhead cost measured in single digit percentages (or less). Only minimal hardware support is required; a single free running counter for the entire machine is sufficient. An alternative that avoids contention is to provide each processor with its own counter; the facility does not require these counters to be synchronized. Hardware designers should provide this level of support, as the resulting accurate timing functionality is of great benefit to both the operating system and users. A single counter in a page of physical memory that can be mapped into applications (by software) is desirable because it makes low overhead timestamps available to applications as well as the operating system.

Chapter 4

Concurrent and Parallel Programming Models

Requirements for scheduling support of applications are heavily dependent on the applications and their programming models. These requirements directly influence the techniques and methodologies used to schedule processors. Single-threaded applications can be adequately scheduled by existing timesharing schedulers, but multiprocessors must support more advanced (multi-threaded) applications that take advantage of parallelism and concurrency. These applications exhibit a variety of scheduling requirements, ranging from no need for additional support to a requirement for dedicated processors. Insights into these requirements and the scheduling assumptions on which they are based can be gained from a classification of programming models for parallel and concurrent programs.

This chapter surveys and classifies programming models for parallel and concurrent programming to determine their scheduling requirements. The classification is based on where additional concurrency is introduced (operating system and/or application levels) and how it is used. The resulting classes range from *pure parallelism* models that introduce no additional concurrency to *dual concurrency* models that introduce and use concurrency at both the system and application levels. Two areas of scheduling requirements for operating system support emerge from the analysis:

1. Support for synchronization with threads that are not currently executing.

2. Support for dedicating processors to applications or portions thereof.

The first requirement exists only in those models that use more threads than processors, while the second is present to some degree in all classes of models.

4.1 Programming Models

Precisely definitions of concurrency and parallelism are a prerequisite to the desired classification of programming models. The *application parallelism* for an application on a multiprocessor is defined here as the actual degree of parallel execution achieved. The corresponding definition of *application concurrency* is the maximum degree of parallel execution that could be achieved if an unlimited number of processors were available. For example, an application consisting of ten independent processes running on a six processor multiprocessor has an application parallelism of six based on the six processors, and an application concurrency of ten because it could use up to ten processors. Application concurrency beyond the hardware parallelism can improve hardware utilization by

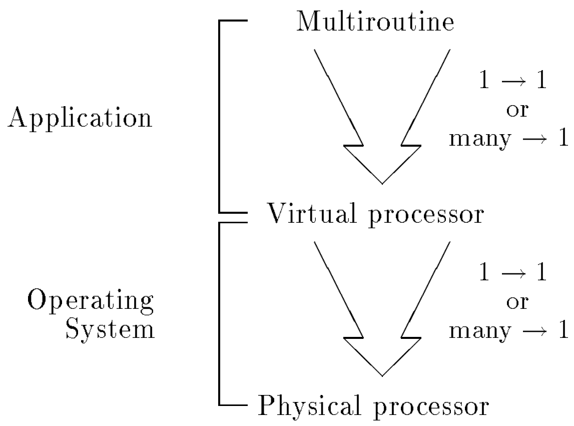

Figure 4.1: Two Levels of Concurrency

Relationship	Model
MR = VP = PP	Pure Parallelism
MR $\geq$ VP = PP	User Concurrency
MR = VP $\geq$ PP	System Concurrency
MR $\geq$ VP $\geq$ PP	Dual Concurrency

Table 4.1: Programming Models for Parallel and Concurrent Programming

allowing other portions of the application to proceed if one portion blocks for some reason, such as a disk or network operation. The use of concurrency can also simplify programming of concurrent applications by allowing the state of ongoing interactions to be captured in the local variables of executing entities instead of a state table.

Programming models for applications can introduce concurrency beyond the hardware parallelism at two levels as shown in Figure 4.1. An operating system can introduce concurrency by providing independently schedulable entities such as Unix processes or Mach threads that it multiplexes onto the hardware processors. The term *virtual processor* or VP is used for these entities. A user level library or language runtime may choose to introduce further concurrency by multiplexing language level entities onto these VPs. These entities are referred to as *multiroutines*; they may be thought of as multiprocessor generalizations of coroutines. Special cases of multiroutines include coroutines (only one VP), and the common programming notion of multiple threads (one VP per multiroutine) as found in the Mach Cthreads library[12]. Multiroutines and virtual processors can be identified in any parallel programming language implementation or environment. An Ada[1] runtime on Unix would employ Unix processes as its virtual processors and Ada tasks as its multiroutines. The Mach Cthreads library [12] employs Mach threads as its virtual processors and Cthreads as its multiroutines.

Programming models for parallel and concurrent programming can be classified based on the relationships among the numbers of multiroutines (MR), virtual processors (VP), and physical processors (PP) supported by the model (Table 4.1). For *pure parallelism* models, the programmer's

[1]Ada is a trademark of the US Government.

notion of concurrency is identical to the hardware parallelism. Compiler generated code for parallel execution of loop bodies is a common example. *User concurrency* models introduce additional concurrency at the user level. An example is programs based on a queue of user-defined tasks that are dequeued and executed in parallel by virtual processors. A *coroutine* model is a user concurrency model with exactly one physical processor and hence, one virtual processor. The *system concurrency* models introduce additional concurrency only at the system level. This class of model is used by most multithreading packages such as parallel language runtime systems including the Mach Cthreads library[12] and the Topaz system developed at Digital Equipment's System Research Center [4]. Finally, the relatively new class of *dual concurrency* models introduces concurrency at both the system and user levels. One example of this is the Portable Common Runtime for Cedar developed at Xerox PARC[54]. The Unix implementation of this runtime limits the number of active virtual processors while allowing large numbers of multiroutines. The runtime converts most blocking operating system operations to their non-blocking equivalents and blocks the multiroutine in the runtime system. Additional virtual processors are available for situations in which operations that block in the operating system kernel cannot be avoided.

The programming model for an application depends both on the programming language or library and how it is used. For example, pure parallelism applications can be written using a library or language with a system concurrency model by creating exactly as many virtual processors as physical processors. If multiple levels of languages and libraries are used, it is possible to introduce more than one level of user concurrency. The above classification does not distinguish among levels of concurrency at the user level, so such models would be classified as user concurrency or dual concurrency models depending on whether they use system concurrency.

4.2 Scheduling Requirements

Programming models that do not use system concurrency usually require dedicated use of processors. These models are required by fine grain applications where the typical length of computation between interactions with other multiroutines is tens to hundreds of instructions. The absence of system concurrency means that every virtual processor is always executing. This assumption is incorporated into the design of synchronization primitives for these models (e.g., spin locks); performance suffers if it is violated. As a result, programs that use these models require dedicated physical processors. The only exception to this is coroutine models because they use at most one physical processor. The major disadvantage of these classes of models is that they handle blocking system operations and page faults inefficiently because blocking a virtual processor in the operating system also blocks a physical processor. This results in wasted time when the physical processor is dedicated to the application.

The remaining classes of models exhibit different scheduling requirements. These classes of models provide efficient support for blocking operations, but potentially large synchronization overheads make them inappropriate for fine grain applications. Blocking operations are efficient because system concurrency can make an additional virtual processor available to utilize the time relinquished by a blocked virtual processor. The blocking nature of synchronization in languages such as Ada, and programming paradigms like message passing, forces the use of models from these classes. The need for dedicated processors in these models is application-dependent rather than being inherent in the model. Applications for which parallel execution is important may need dedicated processors, while others may not. Communication or synchronization with virtual processors that are not running can be expensive because the operating system may not understand which virtual processors are involved. Operating system support for such synchronization and communication can improve performance. Even with this support, synchronization can still consume tens to hundreds of instructions, making these models inappropriate for fine grain parallel applications.

4.3 Conclusion

In summary, there are two areas of scheduling support required by parallel and concurrent programs:

1. Dedicating processors for the sole use of an application.

2. Scheduling support for communication and synchronization among virtual processors for applications that use system concurrency.

The following chapters take up the issues involved in providing this support and describe new Mach scheduling features that support parallel and concurrent programs.

Chapter 5

Scheduling Support for Concurrency

This chapter considers how an operating system scheduler can support concurrent programming and introduces new features of the Mach scheduler that provide support in this area. For a concurrent program with more threads than processors, communication or synchronization with a thread that is not running can be a problem. The scheduler must be convinced to run the other thread, but in many cases there are no effective mechanisms to do this. New mechanisms are presented that can accomplish the desired effect, Discouragement and Handoff Scheduling. Discouragement Scheduling enables an application to inform the scheduler that a thread is currently unable to make further progress, allowing the scheduler to run another thread. Handoff Scheduling goes a step further by naming the other thread that should be run. Initial results indicate that these techniques can improve the performance of both synchronization and communication in concurrent programs.

5.1 Overview

Scheduling support for communication and synchronization depends on the communication and synchronization techniques to be supported. These techniques can be divided into two classes depending on whether they are implemented primarily in the operating system kernel or applications (with support from the operating system). Synchronization can be implemented via semaphores supported by the operating system, or by applications in shared memory with operating system support for blocking and unblocking threads as needed. Similarly, communication can also be implemented by message passing in the operating system or by a shared memory mechanism. Any communication mechanism that can block threads waiting for communication events subsumes the functions of synchronization, but at a potential cost in efficiency. Mach provides a message passing communication facility in the kernel, but expects that the primary implementation of synchronization for applications that share memory will be in the applications (for efficiency).

Applications that use more virtual processors than physical processors can encounter performance problems when synchronizing with a virtual processor that is not running on a physical processor at that time. These problems are caused by the need to run the target thread to complete the synchronization. Currently available tools are often crude: the initiating thread may spin, spin with a context switch, or block. In all three cases, the scheduler does not know which thread is the target thread and may run other threads instead. The goal of this research is to make information available to the scheduler about which thread or threads should be run.

5.2 Discouragement and Handoff Scheduling

These techniques for improving synchronization behavior are based on hints. The Mach scheduler is implemented in the operating system kernel, but allows users to provide hints that influence its decisions. These hints consist of local information involving the thread that provides the hint and possibly one other thread so that users can avoid maintaining information on the overall state of their applications. The hints are based on two pieces of scheduling information that may be available when a thread attempts to communicate or synchronize with another thread that is not running: the first thread may be unable to make further progress until the communication or synchronization is complete, and it may know the identity of the thread it is trying to communicate or synchronize with. For example, in a synchronization based on a message exchange, the initiating thread must block and the identity of the thread that must complete the synchronization is often known.

The Mach scheduler has been enhanced to accept and use hints that provide this information. There are two classes of hints:

1. **Discouragement:** Hint that the current thread should not run. There are three levels:
   (a) Mild: Give up the processor to any other thread if possible.
   (b) Strong: Give up the processor if possible and temporarily depress priority.
   (c) Absolute: Block for a specified time period.
2. **Handoff:** Hint that a specific thread should run instead of the current one.

A new primitive, **thread_switch**, allows hints from both classes to be provided simultaneously. It is documented in Appendix D.

Discouragement hints are useful for optimizing shared memory synchronization in applications that employ system concurrency. The identity of the lock holder is not recorded by the common 'test-and-set' instructions used to implement shared memory locks, making a Handoff hint impossible. A Discouragement hint yields the processor in the hope that the holder of the lock will run. This can cause problems if more than one thread is yielding because they may yield to each other with the result that no useful computation occurs. This can be caused by the timesharing scheduler giving the yielding threads higher usage-based priorities than the thread(s) they are waiting for. Absolute Discouragement prevents this by blocking the threads, but the available time resolution based on clock interrupts is usually too coarse for medium to fine grain synchronization. Strong Discouragement is a compromise that avoids the weaknesses of the other alternatives. A Strong Discouragement hint causes the scheduler to favor threads doing useful work over those that are unable to proceed without the overhead of actually blocking the latter threads. The priority depression caused by Strong Discouragement is canceled when the thread is next chosen to run by the scheduler, or by expiration of a timeout supplied with the hint (whichever comes first). The use of timeouts is necessary to prevent starvation effects.

Handoff Scheduling 'hands off' the processor directly to the specified thread, bypassing the internal scheduler mechanisms. Handoff scheduling may designate a thread within the same task or a different task on the same host to run next. A shared memory lock based on a 'compare-and-swap' instruction can identify the target thread, or it may be available from the structure of an application. For example, a buffer may be empty and exactly one thread performs the task of filling it. One promising use of this technique is to address the problem of 'priority inversion' where a low priority thread holds a lock needed by a high priority thread. The high priority thread can detect this situation and hand off the processor to the low priority thread for the purpose of dropping the lock[1]. When used from outside the kernel, Handoff Scheduling causes the specified thread to be removed from its run queue and run, avoiding a run queue search. Handoff Scheduling is also extensively used by Mach's message passing subsystem inside the kernel to immediately run the recipient of a message. Uses like this inside the kernel gain additional performance because the newly awakened thread can be run immediately without ever placing it on a run queue.

5.3 Performance

This section presents results from two experiments that demonstrate the performance benefits of using scheduling hints. These experiments were performed on a Encore Multimax with NS32332 processors, which have a speed of approximately 2 MIPS. Two experiments were run. The first experiment investigates the use of hints for synchronizing with a thread that is not running. It uses a multithreaded test program than synchronizes with randomly selected threads. A shared variable contains a thread number. That thread replaces it with some other randomly chosen thread number, which replaces it with another randomly chosen thread number, and so on. This program is restricted to run on a single processor, so that it repeatedly synchronizes with a thread that isn't executing. The threads that are not the target of the synchronization use a scheduling hint to encourage the operating system to run the target.

Figure 5.1 shows the elapsed time per synchronization in milliseconds for different scheduling hints and numbers of threads. The error bars are invisible because the standard deviations for all of the results are less than 2% of the measured value. The mild discouragement hint exhibits two different behaviors. If all threads are at the same usage-based priority, the synchronization occurs relatively quickly, as indicated by the Mild-ok curve on the graph. For three or more threads, some of the threads may be at different priorities. This priority inversion causes the synchronization times to increase because multiple high priority threads may be waiting for a low priority thread. Table 5.1 shows the resulting time ranges (Mild-bad); ranges are shown because these results are skewed to higher frequencies at lower values. Multiple runs of the same test cases exhibit both behaviors unpredictably. Longer test runs are more susceptible to priority inversion, so the number of synchronizations per run was reduced for runs involving five or more threads to ensure that both behaviors would be exhibited. The synchronization times from some of the five thread test cases are slightly better than would be expected (based on the other results). This is most likely due to the number of threads being a divisor of the 10Hz clock frequency that drives the scheduler, so that stable behavior is more likely. Absolute Discouragement was not tested because it would result in times on the order of 100ms per synchronization given the hardware clock interrupt frequency of 10Hz.

These results demonstrate the benefits of scheduling hints. Running this program with no scheduling hints yields times of half to one second per synchronization, demonstrating the potential performance penalties for ignoring this problem. Context switching (Mild Discouragement) is effective if there are only two threads in the system, or if all the threads involved are at the same usage-based priority. In general, poor and unpredictable results can be expected from this technique because the timesharing scheduler is continually changing thread priorities. Strong Discouragement has predictable performance, but is slower than the best cases of Mild Discouragement due to the costs of the timeouts associated with the priority depressions. These costs also account for the increasing difference between the 'Strong' and 'Mild-ok' as the number of threads increases. Handoff

[1]Such detection mechanisms usually require support from the implementation of the locking protocol.

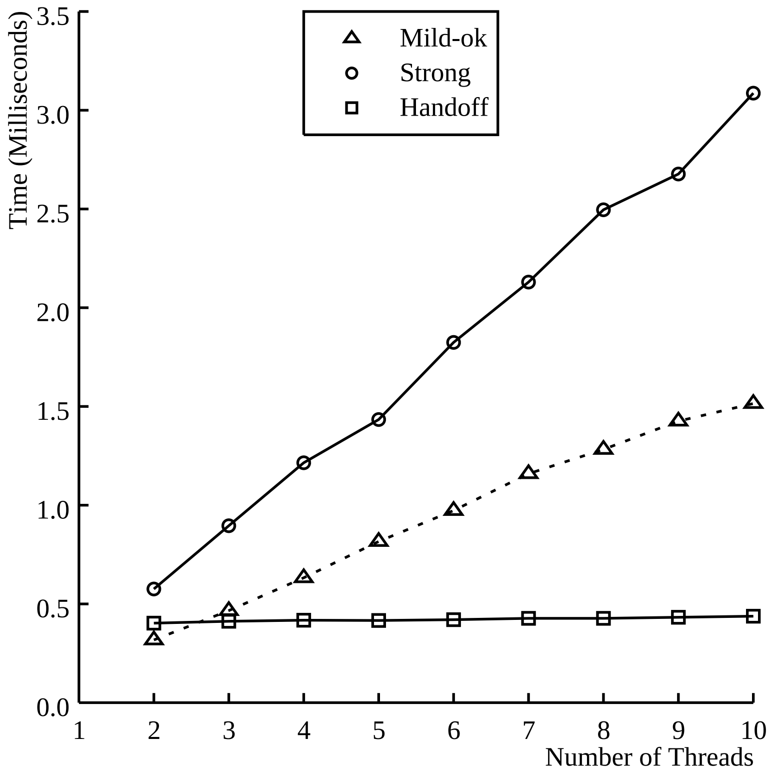

Figure 5.1: Synchronization Experiment Results

Threads	3	4	5	6	7	8	9	10
Min	0.66	1.13	3.88	8	138	264	223	625
Max	1.22	4.43	7.53	607	487	926	572	1173

Table 5.1: Mild-bad Synchronization Times (Milliseconds)

Messages	RPC		1-way	
Idle processor?	No	Yes	No	Yes
No Handoff	1914±11	1630±6	857±5	432±3
Handoff	1848±8	1628±7	861±10	429±4

Table 5.2: Message Passing Handoff Results

Scheduling produces the best performance and is significantly faster than sending a message because no time is spent formatting and transporting the message, or blocking to wait for it. These results suggest that effective support of system concurrent applications requires Strong Discouragement support, and that Handoff Scheduling is a powerful optimization if the information it requires is available. These are worst case results, but are indicative of the relative performance of the hints.

Results reported by Zahorjan, et al[59, 60] confirm the need for mechanisms to improve synchronization with non-running threads. This research investigated the interaction of multiprocessor locks with scheduling disciplines for applications with more threads than processors. An important result is that descheduling a thread that holds a lock can cause significant performance degradation, because 'other threads will spin for a scheduling quantum, a considerable period of time'. Mach and similar operating systems have no knowledge of the locks used by an application, and will occasionally deschedule a thread that holds a lock. The results in Table 5.1 show that Discouragement and Handoff hints are effective remedies for this situation, in part because they can be used to avoid having threads spin for an entire scheduling quantum.

The second experiment concerns the performance benefits of using Handoff Scheduling in the kernel. It uses a message passing exerciser to measure the performance impact of Handoff Scheduling in the Mach IPC system. The experiment involves exchanging messages between two threads in a task on both single and multiple processor configurations. The key difference between the uniprocessor and multiprocessor experiments is the availability of an idle processor to run the recipient thread. Hence, the uniprocessor results are also applicable to multiprocessors when no idle processors are available. Table 5.2 shows the results in microseconds of elapsed time per exchange as mean ± standard deviation. The time differences are statistically significant only for the RPC case without idle processors. The RPC case with idle processors benefits from a handoff in the dispatching code discussed in Section 2.2.2. This handoff was not disabled for these experiments because it is not specific to the IPC system, and because disabling it requires scheduler modifications that affect the critical context switch path. Individual messages do not gain performance from Handoff Scheduling for two reasons: in its absence a sender can queue multiple messages before context switching to the receiver, and on a multiprocessor the sender and receiver can run in parallel with complete overlap.

Based on these results, Mach is configured to use Handoff Scheduling for RPC when no idle processor is available. Current Mach kernels can handoff only once per RPC because the send half of the RPC is implemented separately from the receive. This results in the sender handing off to the receiver before the sender is queued for the reply. When the reply comes back, no thread is queued and no handoff takes place. The Mach IPC system is being redesigned to incorporate a bidirectional message primitive that can handoff in both directions. Similar functionality exists in other systems, such as the Topaz operating system developed at Digital Equipment's Systems Research Center for the Firefly[48].

5.4 Related Work

An important design decision in the area of scheduling support for synchronization is where the scheduler and synchronization logic are implemented. Mach implements the scheduler primarily in the kernel and expects the primary implementation of synchronization logic to be in applications. Alternative approaches to this area include placing both scheduling and synchronization logic in the application, and implementing both in the operating system kernel.

The approach of implementing both the scheduler and synchronization logic in the kernel involves the use of higher level kernel synchronization primitives, such as semaphores or condition variables. The advantages of this approach are that a cleaner interface can be provided by hiding more scheduler details, and higher level primitives can simplify the implementation of a library or language runtime that uses them. The corresponding disadvantages are that languages and libraries must use these primitives to influence the scheduler, and the primitives may be specialized towards some languages or classes of applications. This can affect performance if the primitives are not a good match to the programming language or model. The Topaz system uses this approach, but specialization is not an issue in that environment because most programming is done in Modula-2+[48].

The alternative of implementing both the scheduler and synchronization in user mode is typified by the *policy modules* of systems like Hydra [57] and StarOS [20]. These systems expected the user to make all scheduling policy decisions, and implemented schedulers that would consult application-mode policy modules on a regular basis. Two problems were encountered with this approach: the overhead involved in consulting policy modules, and the complexity of policy modules. The overhead of crossing the application-kernel boundary to consult a policy module made it inefficient to consult such modules for every scheduling decision. As a result, the kernel was forced to implement short-term policy (compromising the separation of policy and mechanism), and was still potentially subject to the basic synchronization problem that motivated this chapter. The complexity of policy modules resulted from an interface that expected the policy module to track the execution state of the entire application and indicate which virtual processors should be running at which priorities. The difficulty and complexity of obtaining and maintaining this information resulted in most policy modules implementing simple suspend/resume functionality that did not take advantage of the available flexibility.

Mach avoids the problems of these systems by using a different design philosophy. The philosophy of *policy mechanism separation* that guided the design of the policy module systems holds that the kernel cannot possibly understand how complex applications synchronize, and that therefore the applications should be making all scheduling decisions for themselves, preempting the kernel's scheduling functionality to prevent problems. Mach's philosophy is that the basic scheduler does a reasonable job much of the time so that it is better to detect and correct mistakes rather than prevent them. This is supported by the observation that many multi-threaded applications run acceptably without any special synchronization support. Mach avoids the overhead problem by implementing the basic scheduler entirely in the kernel, but allowing users to provide hints that influence its behavior. The complexity problem is avoided by designing the user interface to the scheduler to use hints based on local information rather than commands based on global information. This makes it much easier for applications to acquire and provide scheduling information by avoiding the complexity involved in maintaining global state information.

Zahorjan, et al. [59] propose an alternative approach to the problem of descheduling a thread that holds a lock. They propose using a flag to indicate that a thread is holding the lock to prevent this descheduling, but recognize that this has two problems. First, setting and clearing this flag can double the length of the code sequences that acquire and release locks. Second, an unscrupulous user could leave these flags set to obtain better service for his threads. The second problem makes this approach inappropriate for a kernel scheduler that deals with multiple applications, but it could still be used for a single application scheduler (e.g., a user-mode scheduler that schedules multiroutines onto virtual processors). The scheduling hints described here enable the use of a detection and recovery approach instead of this prevention technique. A thread that spins on a lock for too long can infer that the holder has been descheduled and use a scheduling hint to cause it to be rescheduled. The efficiency of this approach is comparable to prevention, and it avoids the overhead and unfair service problems that the latter may cause.

Handoff Scheduling can be further refined for remote procedure calls. Logically, a single thread of control crosses address space boundaries to make the remote procedure call and returns when it is complete. Handoff Scheduling implements this logical single thread by optimizing control transfers between two threads. A further refinement is to use a single thread of control to match the logical control flow [3, 26]. This thread directly executes the call in the remote address space and returns when the call is complete. Performance is improved because only the address space is changed; there is no need to switch threads in the operating system, eliminating the overhead of blocking and waking up the threads involved in the handoff [3]. Hardware support for the address space change can further improve performance of this mechanism by eliminating the trap into and return from the operating system kernel [26]. This technique effectively separates threads from address spaces, and thereby introduces some process control complications. For example, the operation of terminating an address space must be prepared to evict threads that have transferred in from other address spaces (returning an error indication), and prevent the return of threads that have transferred out [3]. This refinement promises to greatly reduce the costs of cross-domain interactions, but further experience is needed to understand the consequences of its separation of threads from address spaces.

Chapter 6

Processor Allocation and Control

A processor allocation facility is an important component of a multiprocessor operating system's scheduler. A requirement for dedicated processors is present (to different degrees) in all four classes of programming models surveyed in Chapter 4, and a simple processor allocation facility, *gang scheduling*, is supported on many multiprocessors. A gang scheduler guarantees to schedule components of an application simultaneously. This is needed by fine grain parallel applications because their performance is severely impaired when any part of the application is not running, but it is also useful for other classes of parallel programs. The overhead involved in assembling a gang of processors for an application can be significant, so the processors are often dedicated for periods longer than the typical timesharing quantum to amortize this overhead. Processor allocation for applications that use system concurrency involves a weaker guarantee than gang scheduling, because not all of the virtual processors can be run simultaneously.

6.1 The Mach Processor Allocation Facility

This chapter describes the design and implementation of a processor allocation facility for Mach. The most important aspect of this design is the division of the facility among three components, the kernel, the application, and a privileged server. This results in a separation of policy and mechanism by isolating the allocation policy in the server so that only the allocation mechanisms have to be implemented in the kernel. This frees the kernel from the impact of policy changes and isolates most of their effects to the server which is much easier to replace, modify, maintain, etc. than the kernel. Support for different multiprocessor architectures is primarily the responsibility of the server (for similar reasons). The kernel interface is designed around an abstraction representing a pool of processors, the *processor set*. This provides support for all classes of applications, including both those that do and do not use system concurrency, and isolates the server from needing to know how the applications are structured. Allocations can be configured to allow applications complete control over which threads execute on which processors at any time without impeding the ability of the system to remove processors from a runaway application.

A simple batch-oriented gang scheduling server has been implemented to validate the design concepts and implementation of this processor allocation facility. This server implements a greedy FCFS policy for granting processor allocation requests subject to fixed time and processor limits. A rollover mode for allocations allows the server to timeslice its portion of the machine among competing applications on a very coarse basis (e.g., a basic time slice of 15 minutes). The server uses two threads that communicate via both shared memory and messages, and has been successfully used by researchers and students in an undergraduate parallel programming course at CMU.

Many extensions to this server's policy are possible, including more sophisticated batch scheduling techniques, and the dynamic adjustment of the number of processors available to each application based on system load. The latter extension works best with applications that can adjust their internal structure to match the number of processors available to them at any time.

6.1.1 Design

The design of Mach's processor allocation facility emphasizes flexibility because Mach supports a multitude of applications, languages, and programming models on a variety of multiprocessor architectures. There are several aspects to this flexibility. It should be possible to allocate processors to applications written in different languages with different programming models. Binding threads to individual processors is not sufficient because applications that use system concurrency need to bind a pool of threads to a pool of processors. This improves their performance by allowing any thread to run on a processor vacated by a blocked thread. The allocation facility should be adaptable to the different multiprocessor architectures that can run Mach, in particular, it should be able to support UMA and NUMA architectures without major changes to the kernel interface. The facility should also accommodate the different administrative policies that are sure to exist at various installations concerning who can allocate how many processors, when, and for how long. Changes to these policies should not require rebuilding the kernel. Finally, it is desirable to offer applications complete control over which threads execute on which processors, but applications that do not want this degree of control should not be forced to implement it.

Mach's processor allocation facility meets these goals by dividing the responsibility for processor allocation among the three components shown in Figure 6.1:

1. Kernel — Allocation mechanisms only.
2. Server — Implements allocation policy.
3. Application — Requests processors from server and uses them. Can control their use if desired.

The server must be privileged to gain direct control over processors. It is also the primary component affected by changes in usage policies or hardware configuration, and is much easier to replace or reconfigure than the kernel. This design assumes that processors will be dedicated to applications for time periods of seconds or more rather than milliseconds to amortize the overhead of crossing the boundaries among the three components to perform allocation. The application to server interface is not specified because it can be expected to change with both policy and hardware architecture. Some servers may require applications to provide authentication information to establish their right to use certain processors, while other servers may require information describing the location of the requested processors in a NUMA architecture. The kernel interface does not change from machine to machine, but some calls return machine-dependent information.

The allocation facility adds two new objects to the Mach kernel interface, the *processor* and the *processor set*. Processor objects correspond to physical processors, and are used to manipulate them. Processor set objects are independent entities to which threads and processors can be assigned, as shown in Figure 6.2. Processors only execute threads assigned to the same processor set and vice-versa, and every processor and thread is always assigned to a processor set. If a processor set has no assigned processors, then threads assigned to it are suspended. Assignments are initialized by an inheritance mechanism. Each task is also assigned to a processor set, but this assignment is only used to initialize the assignment of threads created in that task. In turn, each

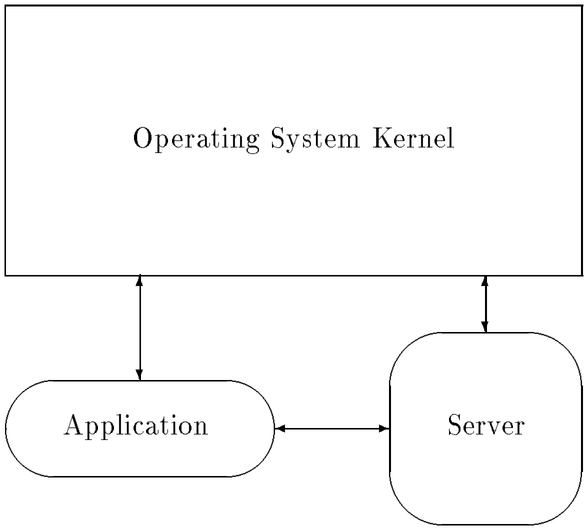

Figure 6.1: Processor Allocation Components

task's initial assignment is inherited from its parent upon creation, and the first task in the system is initially assigned to the *default processor set*. Thus, in the absence of explicit assignments, every thread and task in the system will inherit the first task's assignment to the default processor set. All processors are initially assigned to the default processor set, and at least one processor must always be assigned to it so that internal kernel threads and important daemons can be run. This restriction makes it easier to do processor allocation, because internal kernel threads and important servers always have a place to run. The alternative of allowing all processors to be dedicated to an application would require special support for scheduling these threads and daemons, and would introduce many failure modes in which one or more of these threads and/or daemons was unable to run.

The allocation and use of dedicated processors involves the application, server and kernel. The division of responsibilities is: the server controls the assignment of processors to processor sets, the application controls the assignment of tasks and threads to processor sets, and the kernel does whatever the application and server ask it to do. The following example describes how an application might allocate six processors for its use:

1. **Application** ⇒ **Kernel**: Create processor set.
2. **Application** ⇒ **Server**: Request six processors for processor set.
3. **Application** ⇒ **Kernel**: Assign threads to processor set.
4. **Server** ⇒ **Kernel**: Assign processors to processor set.
5. **Application**: Use processors.
6. **Application** ⇒ **Server**: Finished with processors (Optional).
7. **Server** ⇒ **Kernel**: Reassign processors.

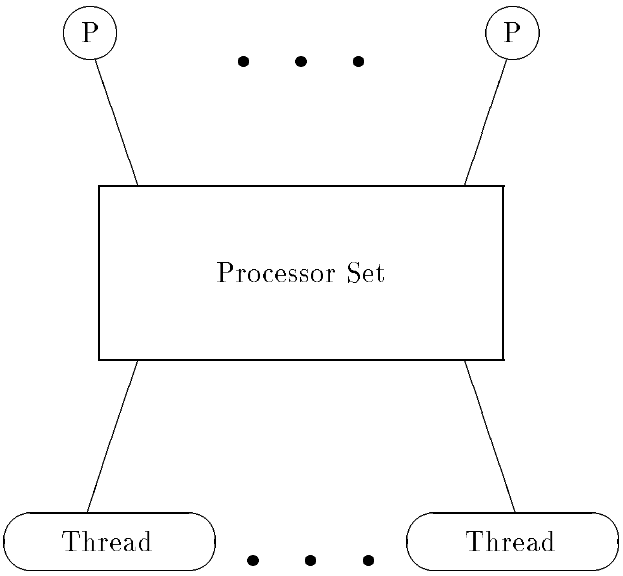

Figure 6.2: Mach Processor Set

This example illustrates three important features of the allocation facility. The first is that the application creates the processor set and uses it as the basis of its communication with the server, freeing the server from dependence on the internal structure of the application. The second is that only one processor set is used. The Mach scheduling algorithms described in Chapter 2 are used within each processor set, and if the task in this example contains six or fewer threads there will be no context switches to shuffle the threads among the allocated processors. The third feature is that the server does not need the application's cooperation to remove processors from it. The server retains complete control over the processors at all times because it retains the access rights to the processor objects. Removing processors without the application's cooperation should not be necessary for well-behaved applications, but it can be useful for removing processors from a runaway application that has exceeded its allocated time.

Protection and authentication for processor allocation is implemented by controlling access to the ports that represent the objects involved. A Mach port is a kernel-protected queue of messages, for which access in controlled by kernel-implemented capabilities, or *port rights*. A task that does not have a port right cannot access the port, even if the task knows about the port via some other mechanism. Port rights can be passed in messages, so tasks that have rights can grant them to other tasks. Ports are used to represent objects implemented by the kernel, so operations on these objects are invoked by sending messages to the corresponding ports. A processor set is represented by two ports, a *name port* and a *control port*. The name port identifies the set and can be used to find out information about it, but all operations on the set itself (e.g., assign threads or processors to it) require the control port. The control port is given to the creator of a processor set. A processor is represented by a single port that allows operations to be invoked on it. These ports will be retained by the privileged scheduling server to ensure its control over the processors. Finally, each host is represented by a *host port* and a *privileged host port*. As in the processor set case, the host port names the host and can be used to find out information about it, but operations that control physical resources on the host (e.g., processor assignment) require the privileged host port. The privileged host port is required to obtain control ports for processors, and can be used to obtain the control port for a processor set from its name port. The privileged host port is currently protected by a Unix authentication mechanism (available only to the super-user). This will be replaced by the kernel supplying this port to the first task on the system which can then implement whatever authentication is appropriate.

There are two different types of privileged ports in the processor allocation facility. The privileged host port represents the privilege to control physical resources on the host. A task that has this privilege (as evidenced by possession of this port) can obtain the ports to control processors. The second type of privilege is control of scheduling in a processor set, represented by the control port for the processor set. This privilege is initially given to the creator of a processor set. For the default processor set, this is part of the privilege to control physical resources because the default processor set always contains at least one processor. As a result, the control port for the default processor set can only be obtained by presenting the privileged host port.

This design meets its flexibility goals. Different programming models and languages are supported by the ability to assign processors to individual processor sets, or to one or more common sets that match the application's requirements. An assignment of one processor to each of a collection of processor sets provides the application with complete control over which threads run on which processors. Isolating scheduling policy in a server makes it easy to change for different hardware architectures and site-specific usage policies. NUMA machines can use this processor allocation facility to match processor sets to clusters of processors with identical memory access characteristics. This disables the kernel scheduler's load balancing between clusters which is a minimum requirement for scheduling on NUMA machines. The allocation facility does not provide a replacement for the disabled load balancing, but the kernel interface has been designed to make sufficient information available for a user-mode implementation of a NUMA load balancer. The kernel interface used by the server and application to perform processor allocation and control are documented in Appendix A.

6.1.2 Implementation

The kernel implementation of processor sets is an extension of the Mach timesharing scheduler described in Chapter 2. The same scheduling algorithms are used within each processor set to avoid unnecessary dependencies on having dedicated processors; applications may still behave differently when assigned to dedicated processors, but the cause of this behavior is to be found in the applications, not the scheduler. The data structure for each processor set contains a run queue that is used as the global run queue for processors assigned to it. The list of idle processors is also maintained on a per processor set basis because a processor can only be dispatched to threads that are assigned to its current processor set. The processor set data structure is also the head of individual linked lists that are linked through the data structures of assigned tasks, threads and processors so that these entities can be found and reassigned when the processor set is terminated. In addition, the data structure contains some state information required to run the timesharing scheduling algorithm, the identities of the ports that represent the set, and a mutual exclusion lock to control access to the data structure.

Processor sets without processors, or empty processor sets, are an important feature of this design, but impair the implementation of the assignment operations. The availability of empty processor sets is an important contributor to isolating the application structure from the server. In the allocation example above, the application assigns its threads to a new processor set before

Operation	Time
Create Processor Set	2250±50
Assign Processor	4772±128
Assign thread (self)	1558±25
Assign thread (other)	2624±185

Table 6.1: Multimax Allocation Operation Performance

the server assigns processors. Empty processor sets also allow the server to do coarse timeslicing of processors among several applications; the applications without processors are in empty processor sets awaiting their turn to run. Threads assigned to empty processor sets are first suspended to ensure that this does not happen to a thread in a non-interruptible wait. This is implemented by suspending the target thread for the duration of the assignment operation and releasing it at completion if the target processor set is not empty; the only exception is the (potentially common) case of a thread changing its own assignment, in which case it will suspend itself at the end of the operation if the target processor set is empty. This suspension also forces the thread to change processors if the target processor set is not empty, as it can only run on processors in that set after the assignment. Processor assignment operations suspend all the threads in a processor set when removing the last processor, and resume them when adding the first processor. The internal implementation of thread assignment is split into two phases, freezing the thread's assignment, and actually changing it. This allows states to be checked and locks acquired by code that relies on the thread's assignment not changing (e.g., to lock a specific thread's processor set, it is necessary that the thread remain assigned to that processor set). It should be emphasized that the suspend and resume operations discussed in this paragraph are the internal hold and release operations; they do not interact with the suspend and resume primitives exported by the kernel.

Table 6.1 reports the times required by some basic operations in the processor allocation system on an Encore Multimax with ns32332 processors (approximately 2 MIPS). The times are shown as mean ± standard deviation in microseconds. The self and other cases of thread assignment correspond to a thread assigning itself and a thread assigning another thread. These times are easily amortized by the expected assignment durations of multiple seconds to multiple minutes. Further optimization work could improve these times if and when support of shorter assignment durations becomes important.

Special techniques are used to manage the processor to processor set assignments. Code in a critical path of the scheduler reads the assignment as part of finding a thread to run on a processor. To optimize this common case against infrequent changes in assignment, each processor is restricted to changing only its own assignment. This eliminates the need for a mutual exclusion lock because a processor looking for a new thread cannot simultaneously change its assignment. The cost to the assignment operation is that it must temporarily bind a thread to the processor while changing the assignment. An internal kernel thread, the *action thread*, is used for this purpose. Current kernels use only one action thread, but are designed to accommodate more than one. The processor assignment interface allows a server to avoid synchronizing with completion of each assignment so that a server thread can exercise the parallelism available from multiple action threads.

Interprocessor interrupts are used for assignment of processors and running threads. Mach's interprocessor interrupt mechanism is designed for generality both in deciding when to send an interprocessor interrupt, and what is done when one is received. Interprocessor interrupts are sent from two places in current Mach kernels. The first is in the scheduler code that queues a bound thread to run when the queuing processor is not the processor that the thread is bound to[1]. The second is thread management code that synchronizes with the pending suspension of a thread that is running on another processor (**thread_dowait**). Processor assignment can cause an interprocessor interrupt to be sent from the scheduler code when the action thread binds itself to the target processor, and thread assignment can cause one to be sent from the thread management code when the thread is suspended as a prelude to changing its assignment. In all cases, the same interrupt is sent, and it causes the invocation of the **ast_check** routine on the target processor to check whether an AST (Asynchronous System Trap) is needed for any reason, and request one if a reason is found. The reasons checked by this routine include: (1) a thread is queued on the local run queue (e.g., the action_thread), (2) the current thread has a pending suspend (e.g., because it is being assigned), and (3) the current thread should be preempted based on its current priority. The third reason allows this mechanism to be used for interprocessor preemption if and when the Mach scheduler supports it.

6.1.3 A Gang Scheduling Server

To demonstrate the utility of this work, a simple processor allocation server for gang scheduling has been implemented. This server is a batch scheduler for processors that allocates processors to requests (subject to limits of 75% of the total processors on the machine and 15 minutes maximum request length). These numbers are based on the usage environment at CMU, and are examples of policy parameters that may vary from site to site. The server satisfies requests in a greedy fashion with strict adherence to the order in which they are received. For example, if the server has 10 processors to allocate and receives requests for 4, 7, and 2 processors, it will satisfy the request for 4 first, and then the requests for 7 and 2 together. The request for 2 processors will not be moved ahead of the request for 7 even though processors are available for the former request. This algorithm was chosen for its simplicity and lack of starvation; more sophisticated algorithms that make better use of the processors by satisfying requests out of order could be used.

The server exports an object-oriented remote procedure call interface to applications. The interface is based on 'request' objects which contain two components:

- A time duration for the allocation request.
- A sequence of <processor set, number of processors> pairs.

A request is satisfied by assigning each processor set its corresponding number of processors for the time duration specified. An application may terminate an allocation before the end of its time duration if it has no further need for the processors. An application interacts with the server by creating a 'request' object, adding ¡processor_set, number of processors¿ request pairs to it, and activating the request. A separate interface routine destroys the request and releases any associated processors for reallocation by the server. Additional interface routines provide information about the server and individual requests. The interface also supports several optional services implemented by the server. The 'destroy' option destroys the processor sets associated with a request when the allocation expires or is terminated. This is useful for novices because it prevents their programs from being suspended when an allocation is exceeded; the programs continue to run, but in the default processor set. The 'notify' option causes the server to send messages to the

[1] These interrupts are not sent to the master processor, due to the frequent use of unparallelized Unix code bound to it.

application after allocating processors and one second before deallocating them. Because processor allocation and deallocation by the server is serial, an application can use these messages to cause barrier synchronizations so that it is never running with less than its full complement of processors. The 'repeat' option causes the request to repeat, allowing the server to timeslice the machine among applications that need more than 15 minutes of time each (using 15 minute timeslices). Finally the 'task' option tells the server that the application is a task with one or more threads; this allows the server to optimize removing processors from the application by performing a task_suspend on it first.

The server is implemented using two threads that communicate via both shared memory and message passing. One thread manages communication with the application clients, the other manages the actual allocation and deallocation of the processors. The primary interaction between these threads is by operations on shared data structures describing the requests, but the interaction thread sends a message to the processor thread when an immediate change to the assignment of processors is needed. One such situation is the activation of an allocation request that can be immediately satisfied.

Library routines are available to hide the server interfaces, so that an application can make a single call indicating how many processors it wants for how many seconds. This routine contacts the server, arranges the allocation, and returns when the server has begun to assign the requested processors. Additional routines incorporate some of the other options available from the server, such as the notify option. The total time taken by this routine is about 35ms to allocate one processor plus the processor assignment time of about 5ms per additional processor (Table 6.1). This overhead is acceptable given the expected allocation durations of tens of seconds to tens of minutes. Both the basic server interface and the library routines that hide it are documented in Appendix C.

The cpu_server and library interfaces have been successfully used by researchers and students in an undergraduate parallel programming course at Carnegie Mellon to facilitate performance measurements of parallel programs. The server removed almost all of the administrative difficulties usually involved in obtaining dedicated machine time. In addition, development of the server repeatedly demonstrated the utility of implementing policy in a separate server because server crashes did not crash the operating system.

Many extensions and changes to the policy implemented in the **cpu_server** are possible. Since it is a batch scheduler for processors, techniques originally developed for batch scheduling of memory, such as assigning higher priority to shorter requests, are applicable. In addition, the server could be extended to allow some users higher or absolute priority in allocating processors, or to allow more processors to be allocated during light usage periods. Finally, the server can be replaced in its entirety by a server that implements a different scheduling policy. One promising new policy is to vary the number of processors available to applications based on the overall demand for processors. A server with this policy can notify applications to reconfigure when it changes the number of processors available. Researchers at Stanford are pursuing this approach and have implemented a server for this scheduling policy under Mach with good initial results[51]. The major benefit of using Mach's processor allocation facility for this policy is that it protects cooperative applications (that reduce their demand for processors when the load on the system rises), from uncooperative applications (that do not). The strict division of processors into processor sets forces the uncooperative applications to compete against themselves instead of cooperative applications. Zahorjan and McCann [61] have conducted simulation studies of related scheduler policies that could be implemented in a Mach scheduling server.

6.2 Scalability

The scalability of Mach's processor allocation and control facility is an important issue because it is intended to adapt to future (larger) multiprocessor architectures. Scalability is of interest for both the overall system, and its individual pieces (kernel interface, kernel implementation, cpu_server and other servers). The architecture of the multiprocessor being managed is of key importance; if it is not scalable, portions of the system that depend on it will not be scalable either. The overall conclusion of this discussion is that the basic design of the facility (including the kernel interface) is scalable although portions of the current implementation are not to the extent that they assume a non-scalable uniform memory access (UMA) multiprocessor architecture.

A scalable software system requires an underlying scalable hardware architecture. UMA architectures are not scalable; all of their realizations have limitations in terms of the number of processors that can share a memory bus and/or the number of ports on multiported memory modules. To the extent that larger caches are employed to leverage these limited resources, the UMA architectures may behave like NUMA machines from a software standpoint (see Section 6.4 for further discussion). In contrast, most NUMA architectures are scalable, although scaling up their size also scales up the access times to remote memory through their processor-memory interconnects. The upshot is that scalability for a processor allocation and control facility requires that it be capable of managing a NUMA multiprocessor.

The overall design of Mach's processor allocation and control facility is scalable, but portions of the current implementation that depend on a UMA multiprocessor architecture are not. The kernel interface is scalable, as it is object oriented and can support parallel operations on independent objects. The current kernel implementation is not scalable, but can be made scalable with some small changes. In addition to adding more action threads to the kernel, data structures may need to be changed so that different threads manage different pools of processors (this allows the threads' kernel stacks to be bound to specific cluster memories in a NUMA architecture); these changes are easy to make. Outside the kernel, the current cpu_server is designed for and assumes a UMA architecture. This server is not scalable, in part because the allocation policy it implements (all processors treated as part of one large pool) is only appropriate for UMA architectures.

An alternative for NUMA architectures is to implement the processor allocation service as a collection of cooperating servers to decentralize the resource management and obtain a structure in which servers run in proximity (memory access delays) to the processors they manage. A promising structure for such a collection of servers is presented by Feitelson and Rudolph [16], although their proposal to use dedicated processors for the servers is questionable. A software implementation of their server hierarchy and algorithms would be a better match to the small overhead of current and proposed processor allocation techniques.

6.3 Scheduling Priorities and Policies

This section describes the Mach system operations that allow applications to control scheduling priorities and policies for threads. These operations are based on the processor allocation interface because the processor set control port represents the privilege to exercise absolute control over the scheduling of any processors assigned to that set. Safeguards are incorporated in the thread assignment operation to protect processor sets from threads assigned from other processor sets with different scheduling policies and assumptions.

Threads have both a priority and a maximum priority as described in Section 2.2.5. A thread's priority is never greater than the maximum priority and the maximum priority can only be decreased by the normal system operation that sets priorities. The maximum priority can be reset to any value by presenting the control port for the thread's processor set. Since the default processor set's control port is privileged, ordinary users who do not use processor allocation cannot raise their thread priorities above their initial maximum. The thread inherits its initial priority from its task and its initial maximum priority from its initial processor set (the one to which its task is assigned) when the thread is created. Tasks, in turn, inherit their priority from their parents on creation. A processor set's maximum priority serves to initialize any threads created in the processor set, and to defend the processor set against any threads subsequently assigned to it; if any such threads have priorities greater than the processor set's maximum priority, these priorities are reset to the maximum priority (likewise for the threads' maximum priorities). The initial priority of the first task, and the initial maximum priority of any newly created processor set (including the default processor set at boot time) are both set to 12 to correspond to the Unix default maximum priority value of 50.

The Mach kernel also supports the concept of a per-thread scheduling policy. Currently, only timesharing and fixed priorities are supported, but the interface is extensible to support additional policies in the future (e.g., round-robin is appropriate in some circumstances). A processor set contains a mask of allowed policies, but this mask is only enforced on thread creation, assignment, or policy change. This allows a number of fixed priority threads to be created by an application that then resets this mask to prevent the creation of others. The mask must always allow timesharing because threads whose policy fails the mask comparison are reset to timesharing; the alternative of a default policy per processor set is more complicated, and the alternative of setting the thread to some allowed policy is potentially unpredictable. The interface routines that manipulate the mask operate by enabling or disabling one policy at a time. StarOS [25] supported similar functionality for selecting scheduling policies, but selected them on a per-processor rather than a per-thread basis and required them to be statically configured when the system is initialized. This feature of StarOS was not used much. The Mach kernel interface routines that deal with policy and priority are documented in Appendix B.

The fixed priority policy supports fixed priority threads by suppressing the usage adjustments of the timesharing scheduler. Absolute preemption occurs between priority levels, and round-robin (or a close approximation thereto) scheduling occurs within each priority level. Preemption may be delayed by up to a clock interrupt period on a multiprocessor because interprocessor interrupts are not currently used for preemption. The fixed priority policy allows each thread to be given a quantum for use in the round-robin scheduling within a priority level. This quantum is given to the thread every time it begins to run, including resumption after preemption by a higher priority thread. The current implementation of the fixed priority policy imposes some additional overhead because it must keep the run queue hint up to date in order to determine when preemptions are necessary.

6.4 Affinity

The processor allocation work described in this chapter is not intended to address the problem of affinity. The area of affinity is driven by the observation that a thread executing on a processor will accumulate a footprint of working set state in that processor's cache (and translation lookaside buffer, depending on the architecture). If that thread blocks and then resumes, rescheduling it on a processor it previously ran on can be more efficient because it may avoid some cache misses. The thread is then said to have an affinity for that processor.

Squillante and Lazowska[45] have conducted queuing theory analyses to investigate including affinity in scheduling policy. They consider policies of binding threads to individual processors (fixed processor or FP), preferring the last processor a thread ran on (last processor or LP), and more complex policies that attempt to maximize affinity in choosing threads to execute. The FP policy is not a good idea for a timesharing environment because it produces long-lived load imbalances, an aspect not considered in their queuing theory analysis. The more complex policies impose additional computational burdens on the scheduler and cause fairness problems for both load and usage balancing. The LP policy is therefore the best choice from these alternatives. This policy has two components; (1) scheduling a thread that returns from a blocking event on its last processor if that processor is idle (thread preference), and (2) choosing a thread that last ran on the current processor when selecting a thread to run from among threads at the same priority (processor preference).

The Last Processor affinity policy is easy to implement for Mach. The Mach kernel contains optional code to implement thread preference for the processor it last executed on, leaving only the processor preference feature to implement. The thread preference code schedules a newly runnable (e.g., unblocked) thread on the processor it last ran on, but does not delay execution to wait for the corresponding processor to become available. If it is productive to wait in such a situation, then the machine should be treated as a NUMA from a scheduling standpoint (even if its memory architecture is uniform). The statement that it is productive to wait in such a situation implies that short term kernel load balancing can have significant performance penalties, a distinguishing characteristic of NUMA machines from an operational scheduling standpoint. The processor preference code would exercise processor preference for threads that last ran on it only among threads at the same (highest) priority.

There are several topics in the Squillante paper [45] that are not appropriate to Mach. The paper analyzes Mach as if the scheduler used simple round-robin scheduler queues, and makes recommendations for policy changes to incorporate affinity. Mach uses priority-based run queues (see Section 2.2.2), making this analysis not directly applicable. In addition, some of the policy recommendations are questionable. Their recommendation for a policy that uses the local run queues to achieve processor binding under moderate loads (e.g., 4) can lead to severe load imbalances, and the suggestion that this can be ameliorated by cycling all threads through the global run queue on a periodic basis threatens to introduce starvation effects. The overall idea of taking affinity into account is sound, but this particular implementation proposal is not; load balancing for a processor binding policy should be done directly between the primary run queues (the local run queues in this case) to avoid starvation. A randomized periodic load rebalancer is probably a better choice than forcing all threads through a secondary (i.e., the global) run queue. The paper also proposes pausing processors in some cases where work is available to avoid large cache reload costs. The pause time calculation is based on an estimate of the reload costs, which is not readily available from current multiprocessor hardware. Finally, the paper proposes adding an affinity measure to priority calculations, but the proposed mechanism requires good estimates of the footprint sizes – these estimates are also not readily available from current hardware. The latter two proposals should be considered by hardware and software designers for future systems, but are not suitable for incorporation into Mach at the present time.

6.5 Heterogeneity

Mach's processor allocation facility does address some heterogeneity problems. By a heterogeneous multiprocessor, we mean a shared memory multiprocessor with processors that exhibit some important incompatibility (usually different instruction sets) among themselves. We divide such machines into the classes of *incompatible heterogeneity* and *compatible heterogeneity*. Machines exhibiting incompatible heterogeneity have more than one class of processors, but each application can only be executed on processors from a single class (but this class may differ from application to application). A multiprocessor consisting of i386 and i860 processors[2] is an example of incompatible heterogeneity because these two processor classes have completely incompatible instruction sets. Machines exhibiting compatible heterogeneity have some applications that can run on more than one class of processors. A multiprocessor consisting of i386 and i486 processors[3] is an example of compatible heterogeneity. i386 applications can be run on i486 processors, but some i486 applications cannot be run on i386 processors because the i486 contains instructions that are not found in the i386. It is possible to mix these two types of heterogeneity. A multiprocessor that contains i386, i486, and i860 processors would exhibit both compatible and incompatible heterogeneity. The processor allocation facility's ability to divide a multiprocessor into sets of processors supports incompatible heterogeneity by ensuring that threads which can only run on a particular class of processor do only run on processors in that class. On the other hand, it does not support compatible heterogeneity because the boundaries in this case are somewhat porous; some threads can cross them while others cannot. It is the author's view that compatible heterogeneity is an orthogonal issue to processor allocation, so that the data structures that support it would be replicated for each processor set. It will take some experience with such systems to understand whether this is the right design decision, or whether something akin to the search path functionality of the Alliant scheduler (discussed in the next section) is a better choice.

6.6 Related Work

Previous work on *policy mechanism separation* has proposed separating the scheduler into two pieces: mechanisms implemented in the operating system, and policy decisions made by *policy module*, usually placed in user mode as part of the application. As indicated in Chapter 5, this work only considered the problem of scheduling within an application and encountered serious problems in the areas of policy module complexity and overhead. Mach's processor allocation system uses policy mechanism separation to attack a different problem, that of scheduling among applications. It avoids the problems encountered in prior uses because processor allocation decisions are made infrequently enough to effectively amortize the overhead of boundary crossing costs, and because the complex policy implementation resides in a server that is implemented once for a system rather than a module that must be customized to each application. If communication between the server and kernel were more frequent, it might have been necessary to place it inside the kernel to minimize the overhead involved; this was the case for real-time scheduling policy modules in a system described by Tokuda, et al [50].

Another body of related work concerns the area of coscheduling. *Coscheduling* is a multiprocessor scheduling policy that attempts to schedule components of an application at the same time, but makes no guarantees about its success in doing so. This policy was originally proposed for medium grain parallel message passing applications (hundreds to thousands of instructions between interactions) that benefit from coscheduling but can achieve reasonable performance in its absence. The major work on coscheduling was done for the Medusa operating system on Cm*[20]. The implementation involved a matrix whose columns were indexed by processors and whose rows

[2] i386 and i860 are trademarks of Intel Corp.

[3] i486 is a trademark of Intel Corp.

were indexed by time; each single-threaded Medusa task occupied one cell in the matrix. Each periodic clock interrupt would cause the corresponding processor to proceed to the next cell in its column; if that cell was empty, it would search other cells in its column to find a task. To achieve effective coscheduling in this system, the processors must advance to the next row in the table almost simultaneously. Cm* made this possible because it had synchronized clock interrupts; the periodic clock interrupts for each processor were generated by an interrupt source (a line clock) that was phase-locked to the 60Hz power supplied by the local electric company. Each processor would therefore take clock interrupts and proceed to the next row of the matrix at almost the same time. A second characteristic of Cm* that this work depended on was Cm*'s limited-memory NUMA architecture that essentially precluded load balancing. Because processors took interrupts almost simultaneously, they could never look at other columns in the matrix when searching for a task to run, and given Cm*'s architecture there was no reason to ever do so. In contrast, the uniform (UMA) shared memory machines that are our primary interest do need and benefit from short term load balancing among their processors. They do not have synchronized clocks, and this makes the load balancing easier to implement by allowing a shared data structure (e.g., the run queue discussed in Chapter 2) to be used.

The Alliant Concentrix[4] scheduler described by Jacobs[24] is an example of an alternative approach to processor allocation and control. This scheduler supports a fixed number of scheduling classes and uses a *scheduling vector* for each processor to indicate which classes should be searched for work in what order. Each processor cycles through a set of scheduling vectors based on time durations associated with each vector, typically fractions of a second. Processes are assigned to scheduling classes by their characteristics or a system call available to privileged users and applications. This scheduler is oriented towards dividing processors among statically defined classes of applications over short periods of time. It contrasts with the Mach orientation of dedicating processors to applications over longer periods of time. Mach's processor sets can be created dynamically as opposed to the fixed number of scheduling classes. Scheduling servers could be implemented by reserving some scheduling classes for their exclusive use, but the static class and vector definitions appear to restrict the flexibility available in forming sets of processors. The Concentrix scheduler also enforces a more restrictive version of gang scheduling in which a blocking operation by any thread blocks the entire gang. This restricts its use to applications that do not use system concurrency and makes parallel handling of blocking operations such as I/O and page faults all but impossible.

An extended version of Mach developed for IBM's experimental RP3 multiprocessor [40] also supports processor allocation. RP3 is a NUMA multiprocessor with a cluster size of one; each processor has a (fast) local memory, but all memories are accessible to all processors via an interconnection network. The cluster size of one lead the implementors of RP3 Mach to adopt an allocation discipline of binding threads to processors[10]. The kernel interface allows threads to be individually bound to (unspecified) dedicated processors. It also supports an indirect notion of the processor; multiple threads can be bound to a single processor by binding the first thread and then binding the remaining threads to the 'same' processor. Explicit processor identifiers are avoided to allow applications to run on any available processors. This is an example of the use of a processor allocation facility to disable inter-cluster load balancing on a NUMA multiprocessor; for larger cluster sizes, the binding would be to the cluster rather than the individual processor within the cluster. Mach's processor allocation facility also supports this functionality, and a server for a machine like RP3 could implement RP3 Mach's processor allocation interface.

[4] Concentrix is a trademark of Alliant Computer Systems.

Chapter 7

Competitive NUMA Memory Management Algorithms

7.1 Overview

This chapter investigates memory management for shared memory multiprocessors that physically distribute their memory among a number of memories local to each processor or cluster of processors. These are non-uniform memory access, or NUMA, multiprocessor architectures. The particular issue of concern is deciding which local memories should contain which data pages. In the migration problem, we operate under the constraint that a page must be kept in exactly one local memory. In the replication problem, we allow a page to be kept in any subset of the local memories, but do not allow a local memory to drop a page once it has it. Theoretical aspects of these problems are covered here; implementation-related issues are covered in the next chapter, and performance measurements are reported in a separate paper [7].

For interconnection topologies that are complete graphs or trees, efficient *on-line* algorithms for these problems are presented. The migration algorithms also extend to interconnections that are products of these topologies (e.g., a hypercube is a product of simple trees). An on-line algorithm decides how to process each request (which is a read or write request from a processor to a page) without knowing future requests. Our algorithms are also said to be *competitive* because their performance is within a small constant factor of that of any other algorithm, including algorithms that make use of knowledge of future requests.

All of the work presented in this chapter is joint work with Daniel Sleator; first person plural pronouns (we, our) are used to reflect and acknowledge this.

7.2 Introduction

A common design for a large, shared memory multiprocessor system is a network of processors for which each processor or cluster of processors has its own local memory [14, 40, 56]. In such a design, a virtual memory system supports a programming abstraction of memory as a single address space without restrictions on how the pages of this address space are distributed among the local memories. A page of this abstract memory may be stored in just one local memory, or identical copies of it may be replicated in many local memories. When a processor p needs to read a location l in a page b, it first looks to see if b is in its own local memory. If it is, the access is accomplished locally. If it is not, a request is transmitted over the network to a local memory containing the desired page, and the contents of l are transmitted back to p.

To clarify the trade-offs involved, it is helpful to consider two extreme cases. Suppose that a page b is read very often by all of the processors, but is never written. In this case, the network use is minimized by replicating b in all of the caches. Once this initial cost is incurred, no further network communication is needed. On the other hand, if a page b is repeatedly written by one processor p, then it behooves the system to eventually migrate b to p's cache, after which no network communication is needed. Since communication bandwidth is frequently a bottleneck in such architectures, we have focused our attention on finding residency strategies that attempt to reduce the total interconnection bandwidth used in processing a sequence of requests.

Most multiprocessors do not have broadcast, invalidate, or snooping mechanisms that maintain consistency among multiple copies of a page when writes occur (note that we are considering multiple memory copies, as distinct from multiple cache copies which can be kept consistent). As a result, we must restrict writable pages to a single copy, so the residency problem becomes one of deciding which local memory should contain this copy; we call this the *migration* problem. The corresponding problem for a read-only page is to determine which of the local memories should contain copies of the page. We call this the *replication* problem because this set of local memories that contains copies of the page is monotonically non-decreasing. We separate the issue of reclaiming memory used in this fashion because there are other competing uses for this memory; this is covered in more detail in Chapter 8. If a page is both read-only and writable at different times, we consider each segment (read-only or read/write) of the page's existence to be a separate instance of the corresponding problem.

Karlin et al.[28] considered a related problem of cache residency on bus-based multiprocessors with coherent caches. These problems are generalizations of the ones we consider because multiple copies of writable data are allowed to exist (and are kept consistent), but the authors only considered bus-based interconnections. This corresponds to a network in which the distance between any pair of nodes is the same, that is, a complete graph with uniform distances. They called this problem *general snoopy caching*, and obtained an algorithm for this problem whose performance is within a factor of three of that of any algorithm for any sequence of requests.

An algorithm is said to be *on-line* if makes a decision about how to process a request based only on that request and the ones before it. Karlin et al. describe a general framework in which to analyze on-line algorithms. They called an on-line algorithm A *c-competitive* if there exists a constant a such that for every sequence of requests σ, and every algorithm B (on-line or off) we have:

$$C_A(\sigma) \leq c \cdot C_B(\sigma) + a,$$

where $C_A(\sigma)$ is the cost incurred by algorithm A on σ, and $C_B(\sigma)$ is defined analogously.

The migration problem can now be stated simply, without reference to the motivating multiprocessor. We are given a network (a graph in which each edge has a length). At any time, there is exactly one node of the network that has the page. A sequence of accesses to the page are generated at the nodes of the network. The cost of an access is the distance between the accessing node and the node with the page. In addition to paying for a request, an algorithm is also allowed at any time to move the page from where it is to another node. The cost of a move is m times the distance between the starting and destination nodes, where m is a constant (which roughly corresponds to the size of the page).

In this work, we consider on-line algorithms for this problem with look-ahead zero. Such an algorithm must satisfy a new request in the current state, and after satisfying the request it is allowed to change its state by moving the page. (This is in contrast to a look-ahead one algorithm, which would be allowed to see the request, move the page, and finally satisfy the request. Look-ahead zero is more natural in this setting. Section 7.5 discusses look-ahead one versions of our algorithms.)

The networks that we consider are those in which the distances are symmetric and satisfy the triangle inequality. If the actual physical network being analyzed does not have a link between every pair of nodes, then the distance between them is the shortest path length in the network between the two nodes.

We have obtained 3-competitive algorithms for the migration problem on the complete network with uniform distances, on any network whose distance metric is that of a tree, and on any network that is the product of trees and/or complete graphs (e.g., a hypercube). A result obtained by Karlin et al. [28] shows that these algorithms are *strongly competitive* in the sense that 3 is the minimum possible competitive factor.

The migration problem is closely related to the problem of 1-server with excursions, defined (but not studied) by Manasse et al.[38]. The migration problem is a special case of 1-server with excursions obtained by restricting the cost of a move (migration) to a uniform constant (m) times the cost of a remote access. In contrast, the move cost for the general 1-server with excursions problem can be arbitrary. Based on our results, we conjecture that 3-competitive on-line algorithms exist for the migration problem on all topologies with symmetric distance metrics that satisfy the triangle inequality.

Section 7.4 contains our results on migration. We describe algorithm M, a 3-competitive algorithm for migration on a network of uniform distances and algorithm M-Tree, a 3-competitive algorithm for the migration problem on any network with the distance metric of a tree. We also consider a specialized version of algorithm M-Tree (called M-UTree) for networks whose distance metric is that of a tree with edges of length 1; the algorithm uses fewer counters than M-Tree, but is only 4-competitive.

We analyze our replication algorithms slightly differently from our migration algorithms. Strictly speaking, the trivial algorithm that initially replicates the page to all of the nodes is 0-competitive for the problem, since we can put all of the cost of the initial page movement into the constant a in the definition of competitiveness. To give meaningful results for this problem, we therefore redefine c-competitiveness to mean satisfying the above inequality with an additive constant of zero.

Section 7.3 contains our results on replication. We give 2-competitive algorithms for the replication problem on uniform networks, and networks with the distance metric of a tree. We also give a specialized version for trees in which each edge has length one that uses less state than the general tree algorithm. These algorithms are strongly competitive, as it is fairly easy to show that the lower bound on the competitive factor for this problem is 2.

In Section 7.6 we describe in more detail how our idealized models for migration and replication relate to real multiprocessor systems.

We consider two topologies for our algorithms:

Complete: A complete graph in which every pair of distinct nodes are separated by the same distance.

Tree: A tree in which distance is additive (i.e., there is a unique path between any two nodes and the access cost is the sum of the access costs for the individual edges along that path).

We also consider a variant of **Tree**, **UTree** in which all single edge access costs are identical (i.e. the cost of an access is the number of edges multiplied by a fixed constant d); simpler algorithms exist given this cost assumption. Our migration algorithms also extend to products of these graphs by employing an independent instance of the appropriate algorithm in each dimension of the product graph; important examples of such products are hypercubes and meshes, which are products of linear trees.

7.3 Replication

We can now give the general abstract form of the replication problem. A set of n nodes, and a distance metric δ_{ij} that specifies the distance between all pairs of nodes i and j are given. In this work, we will only be concerned with distance metrics that are symmetric and satisfy the triangle inequality. A graph G with lengths on its edges is said to satisfy the distance metric δ_{ij} if the shortest path in G between i and j is δ_{ij}.

The general state of the system is described by a bit vector with one bit per node. A node whose bit is 1 is said to 'have the page'. In the initial state of the system, there is a particular single node (which we will always call s) that has the page. As time goes on, the bits of other nodes change to 1. When this happens to a node, it is said to have 'replicated the page'. Once a node has a copy of the page, it retains it.

A sequence of requests to nodes is to be satisfied. The cost of satisfying a request is the distance from the requested node to the nearest node with the page. The cost of replicating the page is r times the distance to the nearest node with the page.

The replication problem is to decide (in an on-line fashion) which nodes should have the page, and to do this in a way that has low cost. As usual, we will compare the performance of a prospective on-line algorithm to that of the off-line optimum on the same sequence of requests. We will assume that the on- and off-line algorithms start in the same state, the one in which the page is in node s.

It is easy to see that no on-line algorithm (for any non-trivial version of this problem) can achieve a cost that is less than twice the optimum on all sequences of requests. Let s and t be the two nodes. Consider the sequence of requests that accesses t repeatedly until the on-line algorithm has given node t the page. Let k be the number of requests in this sequence. The cost incurred by the on-line algorithm is $(k + r)\delta_{st}$. If $k \geq r$ then the optimum off-line algorithm replicates the page immediately and incurs a cost of $r\delta_{st}$. If $k < r$, then the optimum off-line algorithm never replicates the page and incurs a cost of $k\delta_{st}$. In either case, the off-line algorithm incurs a cost that is, at most, half of the on-line algorithm.

Our goal is thus to find on-line algorithms that achieve this factor of two for various distance metrics. We have done this for the cases in which the metric is that of a tree, and the case in which all distances are equal.

It is possible to prove that for certain other metrics (for example, when the graph corresponding to the metric is a four-node cycle), the best competitive factor that an on-line algorithm can hope for is 5/2. We leave these questions to future research.

7.3.1 Replication for Two Nodes

There is a very simple algorithm to achieve the factor of two when there are just two nodes. It is easy to derive the algorithm (which we call **C**) from the lower bound proven above. Let s and t be the two nodes. Algorithm **C** keeps a count of the number of requests to t. When this reaches r, the page is replicated into t. Like the lower bound, the proof that this algorithm is within a factor of two of optimum breaks into two cases. Let k be the number of requests in the sequence. The cost incurred by algorithm **C** is $k\delta_{st}$ if $k < r$ and $2r\delta_{st}$ otherwise. In the former case, the algorithm's performance is optimum, in the latter, its performance is within a factor of two of the minimum cost, $r\delta_{st}$.

7.3.2 Replication for the Uniform Problem

It is very easy to generalize algorithm **C** to many nodes in the case in which the distance between every pair of nodes is the same.

> **Algorithm R:** Maintain a count on each node other than s. The count on a node is incremented each time that node is accessed. When the count on a node i reaches r, the page is replicated into i.

To analyze this algorithm, we partition the cost of a sequence of requests into the costs incurred by the requests to each vertex. The costs incurred by a vertex include the cost of the accesses to that vertex, and the cost of replicating the page there. Each vertex represents an entirely separate two vertex problem. Algorithm **R** is merely running algorithm **C** on each of these separate problems. Therefore, its performance is within a factor of two of optimum.

This analysis is also applicable to *star-shaped graphs*, which are graphs with a central node s such that the distance from any node i to any other node j is not less than the distance from i to s. Hence Algorithm **R** is also within a factor of two of optimum for such graphs.

7.3.3 Replication for Trees

Another easy generalization of algorithm C is to the case in which the distance metric is a tree.

> **Algorithm R-Tree:** The algorithm maintains a count (initially zero) on every node. When a node i that does not have the page is accessed, the count of every node along the path from i to the closest node with the page is incremented. The page is replicated to all nodes whose counts reach r after the access. (Of course, it is not necessary to maintain counts on nodes that have the page. We have expressed the algorithm in this way to simplify the following exposition.)

This algorithm is also within a factor of two of the optimum off-line algorithm. Before we can prove this we need the following observation: The counts on the nodes of a path from s to any other vertex are monotonically non-increasing. This fact is initially true, and is easy to prove by induction. It is also easy to prove that after an access, the nodes with the page are exactly those with counts of r or more.

Consider any algorithm A (off-line or on-line) for this problem. We can assume, without loss of generality, that if algorithm A arranges things so that node i has the page, then all the nodes on the path from s to i also have the page. We can make this assumption because if the algorithm does not do this replication, then it can be modified so that it does the replication, and incurs no more cost. Thus, for any algorithm A, we can assume that the nodes with the page are a connected component in the tree.

These constraints allow us to analyze algorithm **R-Tree** by partitioning the costs incurred by it and by A into parts corresponding to the edges of the tree. An edge incurs a cost for an access operation (equal to the length of the edge), if the path from the accessed node to the closest node that has the page passes through the edge. Otherwise, the cost incurred by the edge is zero. The edge also incurs the cost of a replication across it.

We can view the behavior of algorithm **R-Tree** and any other algorithm A from the perspective of a particular edge. With respect to the game being played on this edge, algorithm **R-Tree** is doing exactly what algorithm **C** would do: when the count on the end without the page reaches r, the page is replicated across the edge. The total cost incurred by an algorithm is just the sum of the costs incurred by all the edges. For each edge, algorithm **R-Tree** is within a factor of two of the cost of any other algorithm for that edge. This proves that **R-Tree** is within a factor of two of optimum.

7.3.4 Replication for Uniform Trees

One disadvantage of algorithm **R-Tree** is that it must keep state for every node in the tree, even though a page can only be replicated to adjacent nodes. If the single edge distances in the tree are constant, this state can be collapsed. The resulting algorithm still involves counters for each node, but the counters for nodes that do not have the page and are not adjacent to copies are always zero. This means that counters need only be maintained for nodes that are adjacent to copies of the page. We will call these nodes *boundary nodes*. Since we are starting with exactly one copy of the page, there is always a unique closest boundary node to any non-boundary node that does not have the page. Our algorithm to implement replication using *boundary nodes* is:

> **Algorithm R-UTree:** Initialize the counters c_i to zero. The algorithm processes a request from a node that does not have the page as follows: find the path to the closest copy of the page, and add the length of the path to the counter for the boundary node on the path. If that counter is $\geq r$, replicate the page into the boundary node, and zero that node's counter. If the value before replication was $> r$, set the counter for the new boundary node on the path to the original boundary node's counter value less r; if this value is $\geq r$, the algorithm loops back to replicate the page into the new boundary node. If the request originated at the original boundary node, then the original counter was $r-1$ before the request, and there is no excess value to be assigned.

The following theorem establishes that algorithm **R-UTree** is strongly competitive.

Theorem 1 *For any sequence σ of requests for the tree page replication problem with constant single edge access costs and any on-line or off-line algorithm $\mathcal{A}$*

$$\mathrm{C}_{\textbf{R-UTree}}(\sigma) \leq 2 \cdot \mathrm{C}_{\mathcal{A}}(\sigma)$$

under the assumption that $\mathcal{A}$ and **R-UTree** *start in the same state with a single copy of each page.*

Proof: Assume, without loss of generality, that all single edge distances in the tree are 1. We merge the actions taken by the two algorithms into a single sequence of events tagged to indicate the algorithms involved. We will give a non-negative (initially zero) potential Φ, such that the following inequality is satisfied by every event:

$$2 \cdot \Delta\mathrm{C}_{\mathcal{A}} - \Delta\mathrm{C}_{\textbf{R-UTree}} \geq \Delta\Phi(t)$$

where the Δ indicates the change in the value of the parameter as a result of the event. Summing this formula over all events and using the fact that the initial potential is no more than the final potential yields the theorem. It remains to specify the potential and verify the above inequality.

Let S be the set of nodes i such that only $\mathcal{A}$ has a copy of the page in node i. We define the potential function as:

$$\Phi(t) = \sum_{i \notin S} c_i + \sum_{i \in S} (2r - c_i)$$

Every step in either algorithm that changes the potential or incurs a cost, results in an event. We now proceed to establish the desired inequality for all possible events.

Consider a replication action performed by $\mathcal{A}$. Let i and j be the source and destination nodes. Since i and j are adjacent, $\delta_{ij} = 1$ by assumption. The cost of the replication to $\mathcal{A}$ is r, so we must show that $\Delta\Phi \leq 2r$. There are two cases to consider based on whether j belongs to S after the replication:

$j \in S$: This means that j does not have a copy of the page under **R-UTree**. j is added to S, so $\Delta\Phi = (2r - c_j) - c_j = 2r - 2c_j \leq 2r$ because $c_j \geq 0$.

$j \notin S$: This means that j already has a copy of the page under **R-UTree**. There is no change to S so $\Delta\Phi = 0$.

Consider a replication action performed by **R-UTree**. This action must be analyzed in combination with the pair of actions that satisfy the request for both algorithms. Let i and j be the source and destination nodes. There are a total of five cases depending on whether j and e are members of S before the replication and on the distance of the access. Let d be the path length of the request; the two simplest cases are for $d = 1$. In this case, $c_j = r - 1$ before the replication, and $c_j = 0$ afterwards. $\Delta C_{\textbf{R-UTree}} = r + 1$ to account for the replication and initial access. The two $d = 1$ cases are:

$j \notin S$: $\Delta C_{\mathcal{A}} = 1$, so we must show $\Delta\Phi \leq 2(1) - (r+1) = 1 - r$. $\Delta\Phi = c'_j - c_j = 0 - (r-1) = 1 - r$.

$j \in S$: This case is free to $\mathcal{A}$, so we must show $\Delta\Phi \leq 2(0) - (r+1) = -(r+1)$. $j \notin S$ after the replication, so $\Delta\Phi = c'_j - (2r - c_j) = 0 - (2r - (r-1)) = -(r+1)$.

The request length, d, is greater than 1 for the remaining cases. Let e be the new boundary node (after replication) for the request that caused the replication. Define x such that $c_j = r - x$ and $c_e = 0$ before the replication where $1 \leq x \leq d$. Then $c'_j = 0$ and $c'_e = d - x$ after the replication, and $\Delta C_{\textbf{R-UTree}} = r + d$. There are three cases depending on whether j and e belong to S before the replication.

$j, e \notin S$: $\mathcal{A}$ has not replicated the page beyond i, so $\Delta C_{\mathcal{A}} \geq d$, and we must show $\Delta\Phi \leq 2d - (r+d) = d - r$. But $\Delta\Phi = (c'_j + c'_e) - (c_j + c_e) = 0 + (d-x) - (r-x) - 0 = d - r$.

$j \in S$, $e \notin S$: $\mathcal{A}$ has replicated the page to j, but not beyond, so $\Delta C_{\mathcal{A}} = d - 1$, and we must show $\Delta\Phi \leq 2(d-1) - (r+d) = d - r - 2$. $\Delta\Phi = (c'_j + c'_e) - ((2r - c_j) + c_e) = 0 + (d-x) - (2r - (r-x)) - 0 = d - x - 2r + r - x = d - r - 2x$. Hence $\Delta\Phi \leq d - r - 2$ because $x \geq 1$.

$j, e \in S$: $\mathcal{A}$ has replicated the page beyond j, so $\Delta C_{\mathcal{A}} = d - k$ where $2 \leq k \leq d$, and we must show $\Delta\Phi \leq 2(d-k) - (r+d) = d - r - 2k$. $\Delta\Phi = (c'_j + (2r - c'_e)) - ((2r - c_j) + (2r - c_e)) = 0 + (2r - (d-x)) - (2r - (r-x)) - (2r - 0) = 2r - d + x - 2r + r - x - 2r = -r - d$. Since $k \leq d$, we have $\Delta\Phi = -r - d = d - r - 2d \leq d - r - 2k$ as was to be shown.

A request whose length is greater than r may cause more than one replication. Since these replications occur in sequence, the above analysis can be applied by setting the request length (d) for subsequent replications to the previous length reduced by the minimum length required to cause the previous replication.

The remaining actions involve satisfying the request. We pair off the corresponding local and/or remote supply actions for a single request and deal with them as a pair provided that they were not dealt with in the previous case. Let r and s respectively be the nodes from which $\mathcal{A}$ and **R-UTree** supply the location, and let t be the node to which it is supplied. When needed, let e be the boundary node for **R-UTree** and this request. Then there are three cases to consider:

$r = s$: If $r = s = t$, then there are no cost or potential changes. Otherwise, both algorithms incur costs of δ_{st}, so we must show that $\Delta\Phi \leq \delta_{st}$. c_e increases by δ_{st}. Because both algorithms performed a remote supply, $e \notin S$, so $\Delta\Phi = \delta_{st}$.

$r \neq s$, $\delta_{rt} > \delta_{st}$: $\mathcal{A}$ incurs a cost of δ_{rt}, **R-UTree** incurs a cost of δ_{st}, so we must show that $\Delta\Phi \leq 2\delta_{rt} - \delta_{st}$. But $\Delta\Phi = \delta_{st}$, as in the previous case, so $\Delta\Phi = \delta_{st} = 2\delta_{st} - \delta_{st} < 2\delta_{rt} - \delta_{st}$.

$r \neq s$, $\delta_{rt} < \delta_{st}$: $\mathcal{A}$ incurs a cost of δ_{rt}, **R-UTree** incurs a cost of δ_{st}, so we must show that $\Delta\Phi \leq 2\delta_{rt} - \delta_{st} = \delta_{rt} - \delta_{rs}$. c_e increases by δ_{st}. Because the distance for **R-UTree** (δ_{st}) is larger than the distance for $\mathcal{A}$, it follows that $e \in S$. Hence $\Delta\Phi = -\delta_{st} \leq 2\delta_{rt} - \delta_{st}$.

Q.E.D.

7.4 Migration

In the migration problem, we must maintain exactly one copy of the page in the network, and we must decide on-line where to keep it. As for the replication problem, let the cost of satisfying a request from a node that does not have the page be the distance between the requesting node and the node with the page. If these nodes are i and j, then this distance is denoted by δ_{ij}. The cost of moving the page from i to j is then given by $m\delta_{ij}$.

Before presenting our algorithms, we show that three is the best competitive factor that can be achieved for this problem.

7.4.1 Lower Bound

This section presents our result that the best possible competitive factor for any non-trivial instance (2 or more nodes) of the migration problem is 3. The situation surrounding this result is somewhat unusual because it has been proved in a previous paper [28], although it is not stated there. The reason for this is that the proof of one of the theorems in that paper establishes a more general result than claimed in the statement of the theorem.

The theorem in question is Theorem 3.3, which establishes a lower bound of 3 on the competitive factor for the "on-line block retention problem in a model allowing *Supplythrough* and *Updatethrough*" provided that there are "at least two caches". This problem differs from our migration problem in that the page (cache block) is permitted to be in more than one place at once, and there is a cost to satisfy certain requests (WRITE) locally if the page (cache block) exists in more than one place; these costs represent the overhead of maintaining cache consistency.

The theorem is proved by considering two caches and a single cache block. There are four possible states for the block; in neither cache, unique to the first cache, unique to the second cache, and in both caches. For an arbitrary algorithm A, the proof constructs a sequence σ consisting solely of WRITE requests for the cache block. This sequence is a "worst-possible" sequence for A because every access is costly. An off-line algorithm H is then described which uses lookahead to process σ more efficiently than A. The properties of σ and the design of H permit it to be shown that H's total cost is one third that of A's at infinitely many points in the infinite sequence σ.

The result can be generalized because H only uses the two states in which the block is unique to a single cache. The two node migration problem can be obtained from the two node block retention problem by restricting the class of algorithms to those which always keep the block present in exactly one cache (i.e., the page is always located at exactly one node). This restricts the selection of A to a subclass of the original algorithms. H is in this subclass (it only uses the two states in which the block is unique to a single cache), and hence the theorem holds for the subclass. This establishes the factor of 3 lower bound on the competitive factor for the migration problem. The theorem can be formally stated as:

Theorem 2 *Let $\mathcal{A}$ be any on-line page migration algorithm for a topology with at least two nodes. Then there is an infinite sequence of requests σ such that $\mathrm{C}_{\mathcal{A}}(\sigma(n)) \geq n$, and*

$$\mathrm{C}_{\mathcal{A}}(\sigma(n)) \geq 3 \cdot \mathrm{C}_{\mathbf{opt}}(\sigma(n))$$

for infinitely many values of n, where $\sigma(n)$ denotes the first n requests of σ.

Karlin et al.[28] also proves that there is no best on-line algorithm for this caching problem; a similar theorem can be proved for the migration problem.

7.4.2 Migration on a Complete Graph

Algorithm **M** solves the migration problem on a graph in which the access cost between any pair of nodes is one, and the move cost between any pair of nodes is an integer m. The algorithm maintains an integral count on each node. These counts are initially zero, and always lie in the range $[0, 2m]$. Let c_i denote the count on node i.

> **Algorithm M**: Initialize all counts c_i to zero. A request from vertex v is processed as follows: If vertex v has the page, then the request is free and nothing happens. If vertex v does not have the page and $c_v < 2m$, then increment c_v and decrement some other non-zero count, if there is one. If vertex v does not have the page and $c_v = 2m$, then move the page to vertex v and set c_v to zero.

The following lemma establishes an important invariant satisfied by algorithm **M**.

Lemma 1 $\sum_i c_i \leq 2m$ *after the completion of each operation.*

Proof: By induction. All the counts are initially zero, therefore the initial sum is zero. An operation that increments a counter increments the total sum only if all other counters are zero (else a non-zero counter is decremented, and the sum is unchanged). Attempting to increment a counter whose value is $2m$ resets it (and therefore the sum) to zero. Hence the sum must be $\leq 2m$ after each operation. **Q.E.D.**

This Lemma has three important corollaries:

1. All counter values are bounded by 0 and $2m$.
2. Before the page is moved, the counter in the destination vertex is $2m$, and all other counters are zero.
3. After the page is moved, all of the counters are zero.

The following theorem establishes that algorithm $\mathbf{M}$ is strongly competitive.

Theorem 3 *Algorithm* $\mathbf{M}$ *is strongly 3-competitive for the migration problem. In particular, for any sequence* σ *of requests and any on-line or off-line algorithm* $\mathcal{A}$

$$\mathrm{C}_{\mathbf{M}}(\sigma) \leq 3 \cdot \mathrm{C}_{\mathcal{A}}(\sigma)$$

under the assumption that $\mathcal{A}$ *and* $\mathbf{M}$ *start in the same state.*

Proof: In analyzing the performance of these algorithms during a sequence of requests, we can partition the things that happen into a sequence of *events* of three types: algorithm $\mathbf{M}$ moves the page, algorithm $\mathcal{A}$ moves the page, and a request is satisfied by both algorithms. We will give a non-negative (initially zero) potential function Φ, such that the following inequality is satisfied for every type of event:

$$3 \cdot \Delta\mathrm{C}_{\mathcal{A}} - \Delta\mathrm{C}_{\mathbf{M}} \geq \Delta\Phi$$

The Δ indicates the change in the value of the parameter as a result of the event. Summing this formula over all the events, and using the fact that the initial potential is no more than the final potential gives the theorem. It remains to verify the above inequality.

Let the location of $\mathbf{M}$'s page be s, and the location of $\mathcal{A}$'s page be t. The potential function we will use is:

$$\Phi = \begin{cases} 2\sum_i c_i & \text{if } s = t \\ 3m - c_t + \frac{1}{2}\sum_{i \neq t} c_i & \text{if } s \neq t \end{cases}$$

Consider the event in which algorithm $\mathbf{M}$ moves the page from s to s'. The cost to $\mathbf{M}$ is m, and the cost to $\mathcal{A}$ is 0, so we need to show that $\Delta\Phi \leq -m$. The corollaries above simplify the calculation of $\Delta\Phi$. There are three cases:

$s' = t$: $\Delta\Phi = 2\sum 0 - (3m - 2m + \frac{1}{2}\sum 0) = -m.$

$s = t$: $\Delta\Phi = (3m - 0 + \frac{1}{2}\sum 0) - 2(2m) = -m.$

$s, s' \neq t$: $\Delta\Phi = (3m - 0 + \frac{1}{2}\sum 0) - (3m - 0 + \frac{1}{2}(2m)) = -m.$

Consider the event in which algorithm $\mathcal{A}$ moves the page from t to t'. The cost to $\mathbf{M}$ is 0, and the cost to $\mathcal{A}$ is m, so we need to show that $\Delta\Phi \leq 3m$. Again, there are three cases:

$t' = s$: $\Delta\Phi = 2\sum_i c_i - \left(3m - c_t + \frac{1}{2}\sum_{i\neq t} c_i\right) = 3c_t + \frac{3}{2}\sum_{i \neq t} c_i - 3m \leq 6m - 3m \leq 3m$ because the counts are bounded by 0 and $2m$.

$t = s$: $\Delta\Phi = \left(3m - c_{t'} + \frac{1}{2}\sum_{i\neq t'} c_i\right) - 2\sum_i c_i = 3m - 3c_{t'} - \frac{3}{2}\sum_{i\neq t'} c_i \leq 3m$ because the counts are non-negative.

$t, t' \neq s$: $\Delta\Phi = \left(3m - c_{t'} + \frac{1}{2}\sum_{i\neq t'} c_i\right) - \left(3m - c_t + \frac{1}{2}\sum_{i\neq t} c_i\right) = \frac{3}{2}(c_t - c_{t'}) \leq 3m$ because the counts are bounded by 0 and $2m$.

Consider an event that is an access operation. Let r be the requested vertex. If $s = t$, there are two cases:

$r = s = t$: There is no cost, and no change to Φ.

$r \neq s = t$: Both algorithms incur a cost of 1, so we must show $\Delta\Phi \leq 2$. The counter increment always adds 2 to Φ. If another counter is decremented, 2 is subtracted from Φ, so $\Delta\Phi \in \{0, 2\}$.

If $s \neq t$ there are three cases:

$r = s$: The cost to $\mathcal{A}$ is 1 and the cost to **M** is 0, so we must show $\Delta\Phi \leq 3$. If no decrement occurs, $\Delta\Phi = 0$. If the counter c_t is decremented, $\Delta\Phi = 1$. Otherwise, some other counter is decremented, and $\Delta\Phi = -\frac{1}{2}$.

$r = t$: The cost to $\mathcal{A}$ is 0 and the cost to **M** is 1, so we must show $\Delta\Phi \leq -1$. The counter increment always subtracts 1 from Φ. If another counter is decremented, then $\Delta\Phi = -\frac{3}{2}$. Otherwise, $\Delta\Phi = -1$.

$r \neq s, t$: The cost to $\mathcal{A}$ is 1 and the cost to **M** is 1, so we must show $\Delta\Phi \leq 2$. The counter increment always adds $\frac{1}{2}$ to Φ. If no decrement occurs, then $\Delta\Phi = \frac{1}{2}$. If c_t is decremented, then $\Delta\Phi = \frac{3}{2}$. Else, some other counter is decremented, and $\Delta\Phi = 0$.

This completes the case analysis.

Q.E.D.

7.4.3 Migrations on an Arbitrary Tree

We now consider the migration problem when the distance matrix has the property that it is the metric of a tree. That is, the access cost matrix $\{\delta_{ij}\}$ has the property that there exists a tree T, with lengths on its edges such that the distance between i and j in T is δ_{ij}. We also let m denote the ratio of the move cost to the access cost between any pair of nodes.

Algorithm **M-Tree** is a 3-competitive algorithm for this problem. Like algorithm **M**, this algorithm maintains a count c_i on each vertex i, and the vertices compete for the page by incrementing and decrementing the counts. These counts are initially zero, and always lie in the range $[0, 2m]$. Algorithm **M-Tree** also makes use of the underlying tree T.

> **Algorithm M-Tree:** Initialize all counts c_i to zero. Let s be the vertex with the page. A request from vertex r is processed as follows: If $r = s$, then the access is free, and the algorithm does nothing. Otherwise, the access is accomplished, some counts are incremented, some are decremented, and finally the page may be moved.
>
> Let P be the path in T from s to r. The counts of the vertices of P (except s) are incremented. A *peripheral path* is a maximal path (one that can't be extended) that starts at s, continues with vertices that have non-zero counts (using only edges of T), and deviates from P as soon as it can. The counts of the vertices on a peripheral path but not on P are decremented.
>
> Finally, if any neighbor in T of the vertex with the page has a count of $2m$, then the page is moved to that neighbor, and the count on the new page location is set to zero. This process is repeated until no neighbor of the vertex with the page has a count of $2m$.

It will be convenient to think of the vertices as forming a rooted tree, with the location of the page s being the root. This defines the children and parent of each node. The counts maintained by algorithm **M-Tree** satisfy the following invariants:

- The counter of the vertex with the page is zero.
- The sum of the counters adjacent to s is at most $2m$.
- At a vertex v other than s, the sum of the counts of the children of v is at most c_v.

The proofs of these invariants are similar to those of Lemma 1, and are omitted. These invariants have several corollaries:

- All counter values are bounded by 0 and $2m$.
- When the server is about to be moved from s to s', $c_{s'} = 2m$, and the counts on all vertices in the tree on the s side of edge (s, s') are zero. (In other words, if the path from v to s' passes through s, then $c_v = 0$.)

We now prove that algorithm **M-Tree** is strongly competitive.

Theorem 4 *Let $\mathcal{A}$ be any on-line or off-line algorithm for the migration problem on a tree. For any sequence σ of requests, algorithm* **M-Tree** *satisfies:*

$$\mathrm{C}_{\textbf{M-Tree}}(\sigma) \leq 3 \cdot \mathrm{C}_{\mathcal{A}}(\sigma)$$

under the assumption that $\mathcal{A}$ and **M-Tree** *start in the same state.*

Proof: We again partition what happens into a sequence of three types of events: algorithm **M-Tree** moves the page from a node to its neighbor, algorithm $\mathcal{A}$ moves the page from a node to its neighbor, and a request is satisfied by both algorithms. Again, we will give a non-negative (initially zero) potential function Φ, such that the following inequality is satisfied for every type of event.

$$3 \cdot \Delta \mathrm{C}_{\mathcal{A}} - \Delta \mathrm{C}_{\textbf{M-Tree}} \geq \Delta \Phi$$

Summing this formula over all the events, and using the fact that the initial potential is no more than the final potential gives the theorem.

Let the location of **M-Tree**'s page be s and the location of $\mathcal{A}$'s page be t. Let the path from s to t be Q. Let δ_{ab} be the distance between a and b in the tree, and let $p(v)$ denote the parent of vertex v in the tree rooted at s. The potential function we will use is:

$$\Phi = 3m\delta_{st} + \sum_{i \notin Q} 2c_i \delta_{ip(i)} - \sum_{i \in Q} c_i \delta_{ip(i)}$$

If the event is that $\mathcal{A}$ moves its page from t to t', then there are two cases, depending on whether the move is toward or away from s ($\Rightarrow s$ and $\Leftarrow s$, respectively). Since $\Delta C_{\mathcal{A}} = m\delta_{tt'}$, we need to show that $\Delta\Phi \leq 3m\delta_{tt'}$.

$\Leftarrow s$: The potential undergoes two changes: the coefficient of $c_{t'}$ changes from $2\delta_{tt'}$ to $-\delta_{tt'}$, and $3m\delta_{t',t}$ is added. The net change is thus $(-3c_{t'} + 3m)\delta_{t',t}$. (Here we have made use of the symmetry of δ.) Since $c_{t'}$ is non-negative, this quantity is bounded by $3m\delta_{t',t}$.

$\Rightarrow s$: This is the reverse of that above, and the change in potential is $(3c_t - 3m)\delta_{t't}$. Since c_t is bounded above by $2m$, this change is also bounded by $3m\delta_{t',t}$.

If the event is that **M-Tree** moves its page from s to s', then there are two cases depending on whether the move is toward or away from t. Since $\Delta C_{\mathbf{M-Tree}} = m\delta_{ss'}$, we need to show that $\Delta\Phi \leq -m\delta_{ss'}$.

$\Leftarrow t$: Φ undergoes three changes: $c_{s'}$ is zeroed, its coefficient changes from 2 to -1, and $3m\delta_{ss'}$ is added. The contribution of $c_{s'}$ changes from $4m\delta_{ss'}$ to 0, so the net change in potential is $-m\delta_{ss'}$.

$\Rightarrow t$: Again, Φ is changed in three ways: c_s is zeroed, its coefficient changes from -1 to 2, and $3m\delta_{ss'}$ is subtracted. The contribution of c_s changes from $-2m\delta_{ss'}$ to 0. The net change in potential is again $-m\delta_{ss'}$.

The most complicated part of the analysis deals with the costs of satisfying the requests. Let r be the vertex that is requested. Let T be the tree rooted at vertex s, and let x be the lowest common ancestor of r and t in T. When the request is satisfied, a cost is incurred by algorithm **M-Tree** and also $\mathcal{A}$. We will associate these costs, as well as the change in potential that occurs as a result of the operation, to the vertices. The potential associated with a vertex i is either $2c_i\delta_{ip(i)}$ or $-c_i\delta_{i,p(i)}$ depending on whether i is on the path from s to t. A vertex i that is on the path from r to s (but is not s) gets a cost of $\delta_{i,p(i)}$ for algorithm **M-Tree**. A vertex i that is on the path from r to t gets a cost of $\delta_{i,p(i)}$ for $\mathcal{A}$. No other vertices incur cost. All the costs incurred by either algorithm, and all the potential changes are partitioned among the vertices in this manner.

Let PR be the path from r to x, let PT be the path from t to x, and let PS be the path from x to s. Furthermore, let P be the part of the peripheral path that is disjoint from the path from r to s. We will examine the costs and potential changes incurred by each vertex, and show that it satisfies the inequality $3 \cdot \Delta \mathrm{C}_{\mathcal{A}} - \Delta \mathrm{C}_{\mathbf{M\text{-}Tree}} \geq \Delta\Phi$. There are several cases to consider, depending on where our test vertex i is with respect to PR, PS, PT, and P.

$i \in PR$: c_i is incremented, so $\Delta\Phi = 2\delta_{ip(i)}$. Furthermore, $\Delta \mathrm{C}_{\mathcal{A}} = \Delta \mathrm{C}_{\mathbf{M\text{-}Tree}} = \delta_{ip(i)}$. This verifies the inequality.

$i \in PS$: Again, c_i is incremented, but this time the coefficient in the potential is -1, so $\Delta\Phi = -\delta_{ip(i)}$. Furthermore, $\Delta \mathrm{C}_{\mathcal{A}} = 0$ and $\Delta \mathrm{C}_{\mathbf{M\text{-}Tree}} = \delta_{ip(i)}$, so the inequality is verified.

$i \in PT$: In this case, $\Delta \mathrm{C}_{\mathcal{A}} = \delta_{ip(i)}$ and $\Delta \mathrm{C}_{\mathbf{M\text{-}Tree}} = 0$. The potential could increase by as much as $\delta_{ip(i)}$ if $i \in P$. (If $i \notin P$ then the potential will not change.) The inequality is certainly true in this case.

$i \notin PR \bigcup PS \bigcup PT$: The costs incurred by both algorithms are zero. The potential will decrease by $2\delta_{ip(i)}$ if $i \in P$, otherwise, the potential will not change and the inequality is verified.

It remains only to verify that the potential cannot be negative. Those vertices on that path from s to t contribute a negative amount to the potential. The most negative contribution they could make is $-2m\delta_{st}$. The initial term $3m\delta_{st}$ guarantees that the potential can never be negative.

Q.E.D.

7.4.4 Migration on Uniform Trees

One disadvantage of algorithm **M-Tree** is that it must keep state for every node of the tree, even though the page can only be migrated to adjacent nodes. If the single edge distances in the tree are constant, we can collapse this state. Unfortunately, this collapsing of state disturbs the cost allocation of algorithm **M-Tree**, so instead of the strongly competitive factor of 3, we obtain a competitive factor of 4 for this algorithm. The algorithm still involves counters for every node of the tree, but the counters for nodes that are not adjacent to the copy of the page are always zero. For a tree whose nodes have at most k neighbors, at most k counters need to be maintained. As before, we call the nodes adjacent to the page *boundary nodes*. We also assume, without loss of generality, that all single edge distances in the tree are 1. Our algorithm to solve the migration problem using boundary nodes is:

> **Algorithm M-UTree:** For each page P, initialize the counters c_i to zero. A request from a node that does not have the page is processed as follows: find the path to the page, and add its length to the counter for the boundary node on the path. Subtract as much of this path length as possible from the counters at the other boundary nodes without making any of them negative (i.e., the total of the decrements to the other counters does not exceed the path length, and is as large as possible without making any of the other counters negative). If the counter at the boundary node on the path is $\geq 2m$, migrate the page to the boundary node and zero its counter. If this counter was $> 2m$, set the counter at the new boundary node for the path to the original counter value less $2m$; if this new counter value is $\geq 2m$, the algorithm loops back to migrate the page to this new boundary node. If the request originated at the old boundary node, then the original counter was $2m - 1$ before the request and there is no excess value to be assigned.

Algorithm **M-UTree** maintains the invariant that the sum of the counters for any page at any node is bounded by 0 and $2m$.

The following theorem establishes that algorithm **M-UTree** is competitive with a competitive factor of 4:

Theorem 5 *For any sequence σ of requests for the tree page migration problem with uniform single edge access costs and any on-line or off-line algorithm $\mathcal{A}$*

$$\mathrm{C}_{\textbf{M-UTree}}(\sigma) \leq 4 \cdot \mathrm{C}_{\mathcal{A}}(\sigma)$$

under the assumption that $\mathcal{A}$ and **M-UTree** *start in the same state with a single copy of each page.*

Proof: Assume without loss of generality that all single edge distances in the tree are 1 (i.e. if i and j are adjacent nodes, then $\delta_{ij} = 1$). Merge the actions taken by the two algorithms into a single sequence tagged to indicate which algorithms performed the actions. As before, we will give a non-negative (initially zero) potential Φ, such that the following inequality is satisfied by every event:

$$4 \cdot \Delta\mathrm{C}_{\mathcal{A}} - \Delta\mathrm{C}_{\textbf{M-UTree}} \geq \Delta\Phi,$$

where Δ indicates the change in the quantity due to the event. Summing this formula over all events and using the fact that the initial potential is no more than the final potential gives the theorem. It remains to verify the inequality for all events.

As in the previous proof, let s be the location of **M-UTree**'s page and t be the location of $\mathcal{A}$'s page. Let Q be the path from s to t. Let δ_{ab} be the distance between a and b in the tree and let $p(v)$ denote the parent of node v in the tree rooted at s. The potential function we will use is:

$$\Phi = 3m\delta_{st} + \sum_{i \notin Q} 2c_i\delta_{ip(i)} - \sum_{i \in Q} c_i\delta_{ip(i)}$$

We now proceed to establish the desired inequality for all possible events:

If the event is that $\mathcal{A}$ moves its page from t to t', then there are four possible cases depending on the relative locations of the pages and whether the move is toward or away from s ($\Rightarrow s$ and $\Leftarrow s$ respectively). Since t and t' are adjacent, $\delta_{tt'} = 1$, so $\Delta C_{\mathcal{A}} = m$ and we need to show that $\Delta\Phi \leq 4m$.

$\Leftarrow s,\ s = t$: The coefficient of $c_{t'}$ changes from 2 to -1. In addition, $3m$ is added to Φ because the distance between the pages has increased by 1. Hence, $\Delta\Phi = 3m - 3c_{t'} \leq 4m$ because the counters are non-negative.

$\Rightarrow s,\ s = t'$: The coefficient of c_t changes from -1 to 2. In addition, $3m$ is subtracted from Φ because the distance between the pages has decreased by 1. Hence, $\Delta\Phi = 3c_t - 3m \leq 4m$ because $c_t \leq 2m$.

$\Rightarrow s,\ s \neq t, t'$: $\Delta\Phi = -3m \leq 4m$.

$\Leftarrow s,\ s \neq t, t'$: $\Delta\Phi = 3m \leq 4m$.

If the event is that **M-UTree** moves its page from s to s', then there are three cases. Let e be the new boundary node for the request that caused **M-UTree** to move its page; if there is no such node, then let c_e be zero. For the first two cases, $\Delta C_{\textbf{M-UTree}} = m$, so we must show $\Delta\Phi \leq -m$.

$\Leftarrow t$: $2m$ is subtracted from the sum of $c_{s'}$ and c_e. Since both have coefficients of 2 in Φ, this subtracts $4m$ from Φ. Since the distance between the pages increases by 1, $3m$ is added to Φ, so the net effect is $\Delta\Phi = -m$.

$\Rightarrow t,\ s' \neq t$: $2m$ is subtracted from the sum of $c_{s'}$ and c_e. Since both have coefficients of -1 in Φ, this adds $2m$ to Φ. Since the distance between the pages decreases by 1, $3m$ is subtracted from Φ, so the net effect is $\Delta\Phi = -m$.

$\Rightarrow t,\ s' = t$: This page move must be analyzed in combination with the actions that satisfy the request that caused **M-UTree** to move its page. Let d be the length of the path for this request to s, then $d \geq 1$. Then, $\Delta C_{\mathcal{A}} = d - 1$ because it must perform an access from t which is closer than s. $\Delta C_{\textbf{M-UTree}} = m + d$ to account for both moving the page and the access from s. Let $c_t = 2m - x$ before the move where $1 \leq x \leq d$. Then after the move, $c_e = d - x$ since x of the d distance was needed to cause the move. c_t is zeroed as part of the move. Since its coefficient in Φ is -1 before the move, this adds $2m - x$ to Φ. c_e's coefficient in Φ is 2, so it adds $2d - 2x$ to Φ. Finally, $3m$ is subtracted from Φ because the distance between the pages decreases by 1. Substituting these into the desired inequality, we have

$$4 \cdot \Delta C_{\mathcal{A}} - \Delta C_{\textbf{M-UTree}} \geq \Delta\Phi$$

$$\begin{aligned} 4(d-1)-(m+d) &\geq (2m-x)+(2d-2x)-3m \\ 3d-m-4 &\geq 2d-3x-m \\ d+3x &\geq 4 \end{aligned}$$

Since $d \geq 1$ and $x \geq 1$, the last inequality is always true, and the desired inequality is established for this case.

This leaves the actions that satisfy the requests. Let r be the node that originated the request. There are three cases depending on the relationship of that node to the pages' positions s and t. Let e be the boundary node for algorithm **M-UTree** in all cases.

$s = t$: Both algorithms incur costs of $\delta_{rs} = \delta_{rt}$, so we must show $\Delta\Phi \leq 3\delta_{rt}$. δ_{rs} gets added to c_e. Since c_e's coefficient in Φ is 2, $\Delta\Phi \leq 2\delta_{rt} \leq 3\delta_{rt}$ because any subtractions from other counters decrease Φ.

$\delta_{rs} > \delta_{rt}$: $\Delta C_{\mathcal{A}} = \delta_{rt}$, $\Delta C_{\textbf{M-UTree}} = \delta_{rs} = \delta_{rt} + \delta_{ts}$, hence we must show $\Delta\Phi \leq 3\delta_{rt} - \delta_{ts}$. δ_{rs} gets added to c_e; since e is either t or between s and t, its coefficient in Φ is -1. Hence $\Delta\Phi \leq \delta_{rs} = -\delta_{rt} - \delta_{ts} \leq 3\delta_{rt} - \delta_{ts}$ because any other subtractions decrease Φ.

$\delta_{rs} < \delta_{rt}$: $\Delta C_{\textbf{M-UTree}} = \delta_{rs}$, $\Delta C_{\mathcal{A}} = \delta_{rt} = \delta_{rs} + \delta_{st}$, hence we must show $\Delta\Phi \leq 3\delta_{rs} + 4\delta_{st}$. δ_{rs} is added to c_e; since its coefficient in Φ is 2, this adds $2\delta_{rs}$ to Φ. A further δ_{rs} is added to Φ if this entire amount is subtracted from $c_{e'}$ where e' is the boundary node on the path from s to t. Hence $\Delta\Phi \leq 3\delta_{rs} \leq 3\delta_{rs} + 4\delta_{st}$ since any other subtractions decrease Φ.

This completes the analysis of all possible actions, and therefore proves the theorem. **Q.E.D.**

The disturbance to the cost allocation of **M-Tree** that produces the competitive factor of 4 for **M-UTree** occurs in the final case above; all of the other cases can be carried through for a competitive factor of 3. This disturbance is due precisely to the collapsing of the counters into the boundary node. For **M-Tree**, the decrement to c'_e would be spread out along the path from s to t (between the servers), and any addition to the potential would be matched by additional cost to $\mathcal{A}$, but for **M-UTree** it is possible to subtract more than this distance. This subtraction cannot be offset against $\mathcal{A}$'s costs and hence requires a larger competitive factor.

7.4.5 Decrementation Variants

The policies for decrementing timers can be changed without affecting the competitive properties of the algorithms. The decrements used in the algorithms as stated are the minimum required to obtain the competitive properties; at most, one counter is decremented after a counter increment. More aggressive decrementing can be performed in two ways without affecting the competitive properties:

1. Decrement more than one counter.

2. Decrement after free accesses (to the node with the server).

Both of these variants tend to discourage migration by subtracting more value from the counters than the original algorithms would.

Decrementing more than one counter will tend to avoid moving the page in response to a random access pattern by increasing the strength of accesses required to cause a migration in the presence of an overall random access pattern. At least one counter (or counters on one peripheral path for **M-Tree**) must be decremented, if possible, to preserve the competitive properties of these algorithms, but up to all of the eligible counters (i.e., non-zero and not incremented) may be decremented without destroying these properties. No counter may be decremented twice. For algorithm **M-UTree** this means that, at most, the distance of the access can be subtracted from each counter, and all counters must be non-negative after the decrements. For algorithm **M-Tree**, the parent-child invariant (at a vertex v other than the location of the server, the sum of the counts of the children of v in a tree rooted at the server location is at most c_v) must be maintained by the decrements; an easy way to do this is to decrement at least one child count (if there is a non-zero one) when the parent is decremented. In the proofs, all of these decrements decrease the potential, except for decrements at the location of or on the path to $\mathcal{A}$'s server; in this case, the potential increases are exactly matched by access costs that $\mathcal{A}$ must incur.

Decrementing after local (free) accesses will tend to leave the page situated at a node that is strongly accessing it; without this feature, a weak access pattern from some other node can cause the page to temporarily migrate away from a node that is accessing it strongly. It is not necessary to decrement any counters in response to a local access, but up to all of the non-zero counters may be decremented without destroying the competitive properties of the algorithms. As before, the decrements for algorithm **M-Tree** must maintain the parent-child invariant at all nodes in the tree. In the proofs, all of these decrements decrease the potential, except for decrements at the location of or on the path to $\mathcal{A}$'s server; in this case, the potential increases are exactly matched by access costs that $\mathcal{A}$ must incur.

7.5 Look-Ahead One

All of the algorithms presented in this chapter are look-ahead zero in that they may not look at the next access when making replication or migration decisions. An alternative model is look-ahead one, in which an algorithm may examine the next access but delay satisfying it until after one or more replication or migration actions have been performed. Look-ahead zero is a better match to the behavior of memory accesses in hardware, because it is unreasonable or impossible to delay satisfying a memory access until a page of data is copied between local memories. In contrast, some caching problems (e.g., General Snoopy Caching in [28]) are inherently look-ahead one because the algorithm can choose how to satisfy an access (fetch location remotely or fetch block from remote cache) after seeing it.

Our algorithms and results carry over to the look-ahead one model with minor changes. For replication, the algorithms remain strongly competitive with a competitive factor of two, and replications now occur in response to the first access after a node's counter reaches r. For migration, the lower bound result is weakened; we can only establish a lower bound on the competitive factor of $3(1 - 1/m)$. m is expected to be large, at least several thousand, so we don't consider this to be an important difference in practice. Modifying the algorithms to migrate on the first access to a node after the node's counter reaches $2m$ (if it is not decremented in the interim) yields look-ahead one algorithms with the same competitive factors.

7.6 Applications of the Algorithms

The algorithms we have presented and analyzed are applicable to a significant collection of existing and proposed multiprocessors. Each node in the graphs used by the algorithms corresponds to a processor-memory cluster in a multiprocessor realization of that graph's interconnection topology. The primary hardware requirement for use of these techniques is that the hardware implement a shared memory model (i.e., support access forwarding). This excludes most current implementations of network shared memory on local area networks; in this case, access forwarding is not possible because page faults must be satisfied by data in a processor's local memory. This is also the case for most current hypercubes and related machines, although research has been conducted into similar machines that do support access forwarding [44].

A secondary requirement is that the reference counting information needed by our algorithms be available. There are several potential methods for doing this:

- Chapter 8 proposes a complete hardware implementation of our algorithms for **Complete**.
- Hardware reference counters (per cluster $\times$ per page) could be used in combination with a periodic software scan.
- Holliday [23] describes experiments that employed software-implemented usage counters based on periodic scans of page table reference bits.

In both cases involving experimental data, mixed results have been obtained for these and similar [23] techniques. It is our opinion that software-implemented counters based on page table reference bits are sufficient for replication, but not for all cases of migration (in particular, they are likely to fail to capture cases in which two clusters are actively using a page, but the usage in one cluster is more intensive than the other). We would recommend that multiprocessor architects and designers consider providing per-processor reference counters for some portion of the shared memory subsystem; this would allow implementation of our algorithms and make reference data available for other uses (e.g., hardware performance analysis).

A secondary issue that comes up in the area of reference counters is how references should be counted on machines with caches; in particular, should cache hits be considered. Removing cache hits from the reference counts removes a large amount of locality, but this corresponds exactly to the function of a cache; taking advantage of locality to avoid loading the memory subsystem. Our algorithms apply to reference streams consisting entirely of cache misses and writebacks/writethroughs, so an implementation that counts only those references that reach memory is reasonable. Despite this there are two potential reasons to count cache references:

- At least one proposed research machine exhibits different cache behavior for remote and local pages (remote pages are uncachable) [5]. An implementation on this hardware should count all local references that hit in cache because they would miss if the page were remote.
- Cache hit traffic may be a good predictor of cache miss traffic. This is an open question requiring further study.

It is certainly simpler (from a hardware standpoint) not to count cache hits; this allows the reference counters to be implemented in the memory proper, as opposed to the various caches.

There are many existing and proposed multiprocessors exhibiting the the **Complete** topology that are amenable to our algorithms and techniques. These machines include network-connected NUMA multiprocessors such as the the BBN Butterfly [14] and IBM RP3 [40], as well as bus-based machines such as the Encore Gigamax [56]. Numerous proposed machines such as the NYU Ultracomputer [21], and the directory-based cache machine (DASH) at Stanford [36] would also support our algorithms. In contrast, the **Tree** and **U-Tree** topologies are applicable to far fewer machines. The only existing machine that comes close is the experimental ACE multiprocessor developed by IBM's research division [19]. This machine is based on romp microprocessors with small local memories and a large global memory. The access ratio (local:global:remote) is an inverted triangle inequality in which the third side is longer than the sum of the other two. We have not thoroughly investigated extensions of our algorithms to this case or to the case of tree machines satisfying a triangle inequality (where it is cheaper to cross a node than to stop there and then move on); in both cases, preliminary work has convinced us that the extensions are not straightforward. Tree-based machines using an architecture such as Fat-trees [35] would also be amenable to our techniques.

Our migration algorithms also extend to product topologies; the appropriate algorithm is run independently in each dimension of the resulting topology. The most common examples of such topologies are hypercubes and meshes, which are products of linear trees; our algorithms for **Tree** and **U-Tree** apply to such machines. Scheurich and Dubois have independently discovered our migration algorithm for **U-Tree** and investigated it on a mesh machine; they were not aware of its competitive properties [44].

Using our migration techniques on rings and products involving rings (e.g., tori) is problematic due to cycling and pinning effects. Bidirectional effects exhibit the phenomenon of *pinning* in which accesses in both directions from the far side of the ring can pin a page in place and prevent it from moving. Unidirectional rings or unidirectional routing structures imposed on bidirectional rings avoid pinning, but exhibit the related phenomenon of *cycling* in which a static access pattern distributed over the ring can cause a page to cycle around the ring when it should stay put. These effects would cause the size of the ring to enter into the competitive factor for the straightforward extensions of our algorithms to these topologies; a more sophisticated approach is needed.

7.7 Further Work

The primary problem of interest from a theoretical standpoint is the migration problem (1-server with excursions). This is the first work to be done on that problem, so competitive algorithms for migration on other topologies is an open area for research. The authors have used the techniques developed in [38] to investigate some small graphs (other than those considered in this work) whose distance metrics satisfy the triangle inequality. Our results indicate that 3-competitive algorithms exist for the small examples investigated. Based on our results and experience, we believe the following conjecture to be true:

Conjecture: *There exists a 3-competitive algorithm for the migration problem for any topology having a symmetric distance matrix that satisfies the triangle inequality.*

Another direction for extensions of this work is to consider randomized algorithms for the migration problems. For the randomized model of competitiveness, the on-line algorithm is allowed to make use of random choices. The cost incurred by the randomized algorithm on a sequence of requests is defined to be the average of its costs over all of the possible series of random choices. Competitiveness is defined as before, but it uses this modified definition of cost. For a number of different problems, it has been shown that the competitive factor can be reduced by the use of randomness [17, 27].

7.8 Conclusion

We have presented and analyzed new strongly competitive algorithms for replication and migration problems that arise in the management of distributed shared memory for multiprocessor systems.

These algorithms are applicable to many existing and proposed multiprocessor architectures. The proofs of the competitive properties of the algorithms have also served to establish new results in the area of competitive algorithm analysis for server problems. We have also briefly highlighted some of the issues involved in actually applying these algorithms to real systems.

Chapter 8

Implementing NUMA Memory Management

8.1 Overview

This chapter considers concerns that arise in implementing the algorithms for managing non-uniform shared memory discussed in Chapter 7. Both hardware and software implementation concerns are covered. A case study of the Mach operating system indicates that integration of these algorithms into operating systems does not pose major problems. On the other hand, hardware support is required to obtain the full functionality of the algorithms.

A basic architectural model that corresponds to a number of actual multiprocessors is used. This model is an example of the **Uniform** topology from Chapter 7, and is used to evaluate algorithms **R** (for replication) and **M** (for migration). The model drives both the implementation analysis for Mach and studies on the performance of these algorithms reported elsewhere [7].

8.2 Processor-Memory Model

The basic architectural model used for the evaluation of the algorithms assumes an idealized machine composed of processor-memory clusters, with physical memory divided entirely among the clusters as shown in Figure 8.1. A processor-memory cluster consists of one or more processors with local memory that is equally accessible (in terms of latency) to all processors. This idealized machine has two distinct memory access latencies; the latency to access memory in the same cluster, and a significantly larger latency to access memory in another cluster. As a result, all memory within a single cluster is equivalent, and all processors within a cluster have identical memory access characteristics (latency in terms of the cluster in which the accessed memory is located). Finally, all memory locations outside the cluster have the same access latency from any processor in the cluster.

This basic model subdivides memory into pages and pages into locations. Pages are the fundamental unit of memory management; locations are the fundamental unit of memory access. The existence of virtual memory mapping mechanisms is assumed, and therefore a distinction is made between virtual pages (in the address space of some program or the operating system) and physical pages (actual memory in the clusters). Mapping virtual pages to physical pages is among the responsibilities of a memory management facility. Sharing may result in more than one virtual page in one or more address spaces being mapped to the same physical page. The page size used by

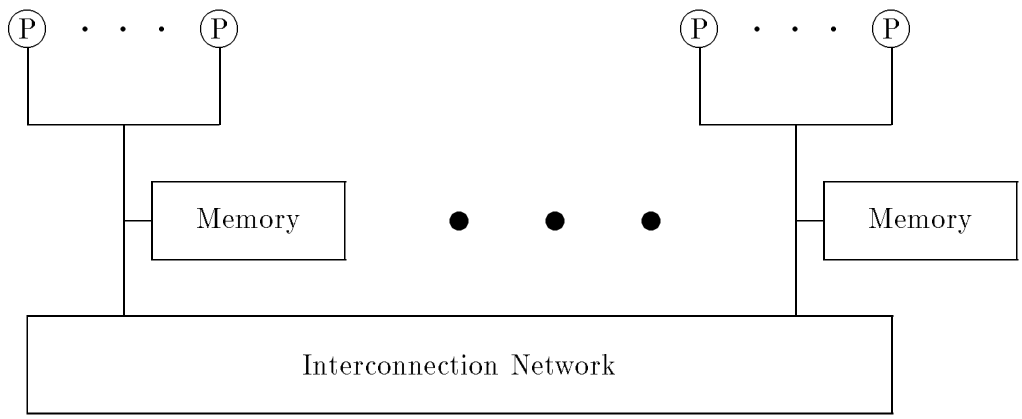

Figure 8.1: Clustered NUMA Architecture

the algorithms can be no smaller than the hardware page size if mapping is used, but it may be a multiple thereof.

The model is normalized by assuming that the difference in cost between an in-cluster memory access and a remote-cluster memory access is 1; this cost includes the effects both of increased latency and use of interconnection bandwidth. This cost only applies to accesses that actually use the interconnection network. In addition, read and write costs are assumed to be identical, but the results generalize to situations in which this is not the case.

This model permits the analysis of techniques for managing the performance impact of distribution in a shared memory system. Two techniques are of particular interest, *replication* and *migration* of virtual memory. Replication consists of making a copy of a virtual page in another cluster and updating mappings that benefit from this copy (in reduced access time). Migration consists of moving a virtual page from one cluster to another and updating all mappings to that page. The costs of replication and migration are formalized as r and m respectively in terms of access costs. These costs include latency and overhead components, but do not include the additional costs of allocating a physical page in a cluster with a page shortage (i.e., causing pageout) or the additional benefits of freeing a physical page in such a cluster (i.e., avoiding pageout). The issues involved in page reclaim are separated from migration and replication and addressed in Section 8.4.1.

The basic model applies to any machine that can implement NUMA memory. This includes NUMA machines that implement the model directly (e.g., Butterfly [14]), NORMA (no remote memory access) machines with uniform access costs, and network shared memory implementations on networks with uniform communication costs. For the last two classes of machines, it is essential that the system (hardware and/or software) support access forwarding so that accesses to pages that are not in local memory can be satisfied at remote memory *without* moving the entire page to local memory (an expensive operation). Most current NORMA machines (e.g., hypercubes) and network shared memory implementations [11, 37, 58] do not support this functionality.

8.3 Basic Problem

The basic problem is management of distributed shared memory in architectures conforming to model. For architectures utilizing a single copy of the operating system (NUMA multiprocessors), this includes not only memory shared explicitly, but also memory shared implicitly via copy-on-write techniques. Because this work relies on replication and migration to perform this management, the problem can be restated as "When and under what circumstances should (virtual) pages be replicated into or migrated to memory in other clusters?"

There is a significant difference between this problem and the related problem of snoopy caching; this model and most of its realizations do not have broadcast, invalidate, or snooping mechanisms that can maintain consistency among multiple copies of a virtual page when writes occur[1]. This prohibits replication of writable pages. Because the issue of page reclamation has been separated, migration of read-only pages make little sense; replication is less costly, and provides the benefits of local access to two clusters instead of one. As a result, the overall problem splits into two sub-problems:

- Replication of read-only pages.
- Migration of writable pages.

If a virtual page is both read-only and writable at different times during the execution of an application, each segment (read-only or read/write) of the page's existence is considered to be a separate instance of the corresponding problem.

8.4 Operating Systems Issues

There are two sets of operating systems issues that must be addressed in implementing our algorithms: (i) how to take into account the limited size of physical memory; and (ii) the interactions between the proposed algorithms and the memory management portion of an operating system. The first issue involves interactions among the various uses of memory (including our algorithms), and in particular, how to allocate memory among these users of memory. The second issue arises primarily if the algorithms are implemented in the operating system kernel; this is an attractive choice both because it permits direct access to mapping information and also because it makes the resulting benefits available to all applications on the system, instead of just those that are modified to use our algorithms.

8.4.1 Limited Physical Memory Size

Since there are many other demands on physical memory besides those generated by replication and migration (e.g. memory allocation, file mapping, internal use by the operating system, etc.), extending the replication and migration algorithms to control memory usage is not appropriate. The operating system should separate reuse of physical memory (pageout or page reclaim) from replication and migration issues. Even the fallback position of dedicating a fixed amount of physical memory to replication/migration is not a good idea; this prevents reallocation of memory to the uses for which it is in greatest demand.

[1]Researchers have recently begun to investigate systems that do incorporate page-level consistency mechanisms; PLUS [6] is one example. The page replication/migration problem for such systems is similar to the cache residency problem for snoopy caches.

The use of independent pageout daemons is an appropriate technique for managing the various cluster memory pools. These daemons can respond appropriately to the potentially different memory demands from cluster to cluster. Any of several standard paging algorithms can be used to implement the daemons [47]. The migration and replication costs can be dynamically modified to feed information about page availability back into the replication and migration algorithms. These modifications should be restricted to increasing costs above their basic levels to reflect page shortages and hence discourage future use of memory in clusters with page shortages. Decreasing migration costs to encourage freeing memory in clusters with shortages, and cost-based reclamation of replicates are fraught with potential danger; this is because not all system components that use memory are or can be sensitive to costs – hence these cost-driven alternatives may result in heavily used pages being evicted in order to retain lightly used ones for cost-insensitive components.

8.4.2 Memory Management Interactions

The replication and migration algorithms can be incorporated into the operating system's memory management code on a NUMA multiprocessor. Implementing these algorithms inside the operating system allows their benefits to accrue to all uses of the machine, but also results in interactions with other memory management functions that must be dealt with as part of the implementation.

The virtual memory management portion of the Mach operating system [41] is used as a base for a case study of these interactions. The Mach VM implementation is cleanly split into machine-dependent and machine-independent portions. The machine-dependent portion consists of a single module, the *pmap* module, that is responsible for all physical map operations. The machine-independent portion of the system associates a pmap with each address space and invokes the pmap module as needed to perform mapping operations. Mach supports parallel execution of multiple threads within a single task's address space; this parallel execution can result in a single pmap being used simultaneously by more than one processor.

Mach envisions support for non-uniform physical memory by adding a NUMA layer between the machine-dependent and machine-independent portions of the VM system [47]. This layer hides the non-uniformity of the memory structure by translating *logical pages* (manipulated by the machine-independent portion of the VM system) to physical pages (in the hardware) in order to implement architecture-specific memory management policies (e.g., replication, migration). A similar translation process is needed for pmaps to allow replication within a single address space if its threads are spread across multiple clusters; in this case, each cluster would have its own physical map, but the collection of these pmaps would appear as a single logical pmap to the machine-independent portion of the VM system. This adds additional complexity to the NUMA layer to better support multi-threaded applications, and may complicate interfaces that allow users to modify replication and migration behavior because an address space no longer uniquely specifies a cluster.

There are two other minor interactions of the NUMA layer with the remainder of the Mach VM system, and one major interaction. The two minor interactions are:

- Pageout functionality must be moved into the NUMA layer and redesigned to use multiple pageout daemons as discussed in Section 8.4.1. The resulting daemons must cope with system-wide (logical) page shortages as well as page shortages in the individual clusters.

- There must be a physical page available for every free logical page. Therefore, use of a physical page for replication may require stealing a logical page from the resident page subsystem to maintain this invariant. Freeing of such a replicate should cause the stolen logical page to be returned to the resident page subsystem's free list.

Neither interaction poses great difficulties for an implementation.

The major interaction involves replication and copy-on-write. If the system has replicated a shared page that must be copied if written, then the replicates can be used to satisfy write faults on the page; this avoids the costs of creating an extra copy, but imposes extra costs if the replicate was used by more than one address space and has to be recreated as a result. The easy case is if there is a replicate that is only being used by the address space that caused the write fault; this replicate can always be used to satisfy the fault. For multiple address spaces, I would propose always using the replicate unless one of the other spaces has indicated that the replicate is needed (cf. the **always replicate** operation in Section 8.7). An additional primitive must be added to Mach's machine-dependent interface to implement this functionality; the fault handler must be able to find out if the NUMA layer has a replicate that can be used to satisfy a write fault.

8.5 Hardware Support

Existing multiprocessor hardware will not allow a sufficiently accurate implementation of the NUMA memory management schemes discussed here. Software systems that impose a level of indirection on all accesses to memory or shared memory can not hope to recover from this performance penalty. Thus it is worth considering an architecture with hardware support for our algorithms. For each page, a set of two reference counters is required per processor in the system. Together with increment/decrement logic, these maintain the counts required by the algorithms. An exception is caused when a pre-loaded threshold for migration or replication is reached. The operating system then deals with the copying and remapping operations required.

The counters are kept with their associated memory page. For a 64-processor system and 16-bit counters, this requires 256 bytes of memory per page. This translates to 50% overhead for 512-byte pages, 25% overhead for 1K pages, 6.25% for 4K pages and 3.125% for 8K pages. This overhead is acceptable for pages in the 4K–8K range. There should be no impact on access times to memory, as the counter manipulations can be performed in parallel with or in a pipeline stage after the actual access. An exception indicating the need for a replication or migration action would occur after the access is complete.

The replication algorithm only needs to increment a single counter. On the other hand, the migration algorithm requires updating two counters, one of which must be chosen from the non-zero counters for that page (local references only update the latter counter). Since the updating of the counters must take place transparently and at the same rate as memory references, a sequential search for non-zero counters is not acceptable. An alternative is to pick a counter and decrement it if it is non-zero. This is much simpler to implement and trace-driven simulations [7] indicate that its performance is similar to the original migration algorithm.

A major alternative to the replication of code is the use of instruction caches. Since code can be assumed to be read-only, the global cache consistency problem does not arise. However, unlike the schemes proposed here, instruction caches do not help with replicatable data.

This architecture has been simulated using traces from a uniform memory access multiprocessor; the results are reported in [7]. The simulated access costs were configured to approximately match a Butterfly, and traces derived from actual parallel programs were used. The results indicate that considerable performance benefits seemed to accrue from both replication and migration, but upon further examination it became clear that most of the benefits were from replication of code and migration of stack pages to the processor using them[2]. Migration effects were observed in shared

[2] Holliday[22] discusses the importance of positioning code, stack, and local pages correctly.

data, but no performance gain was obtained[52]; this could be due to short domains of locality that would not benefit from migration at the page level of granularity, or to the short traces used (each trace represented a fraction of a second of actual computation). In addition, very little effect was observed from varying the page size and thresholds used to make replication and migration decisions. The lack of threshold effect suggests that the worst cases for performance of these algorithms do not occur in practice, so that actual performance is better than indicated by the competitive factor bound. On the other hand, it is possible that these results are also due to short trace effects.

8.6 Replication of Writable Pages

So far we have not allowed the replication of writable pages. For portions of shared memory that are rarely written (called *mostly-read objects* in [53]), the amortized costs of the atomic updates required by the writes may not be prohibitive. Such a scheme can be implemented by using hardware mechanisms to cause a trap if a write occurs to any of the replicates. The handler for this trap can then perform the update by obtaining an exclusive lock on the logical page structure and propagating correct data to all copies. The resulting serialization of writes preserves sequential consistency. Relaxed consistency constraints are preferable if the data has to be updated frequently. On the other hand, if the memory is never written after some point, then replication is a very good idea. Researchers working on the ACE project[19] at IBM Research have found this to be the case for a parallel shortest path program; the data structures describing the graph to be searched are never written after the initialization phase, but are read heavily during the search. Replicating these structures into local memories on their machine produced major improvements in the run time of the application [8]. Similar results have been reported in two recent papers on NUMA memory management [9, 13].

Algorithm **R** may not be appropriate for managing replicated writable shared memory because it ignores the costs of updating other replicates in response to a write. The *General-Snoopy-Caching* algorithm in [28] is a better choice if these costs are important because it takes them into account; this algorithm is strongly competitive with a factor of 3. If update costs depend on the number of replicates (e.g., if individual messages are required to update each replicate), then the algorithm must be modified accordingly in order to remain strongly competitive.

8.7 Input and Feedback

If additional information is available about the access patterns for a page, the algorithms can be further improved upon. Four primitives are proposed to help specify this additional memory usage information:

Never replicate: On average, this page is used so infrequently in this cluster that it should never be replicated (even if it accumulates r accesses).

Always replicate: On average, this page will be used enough in this cluster to justify replication as early as possible. Alternatively, this page is read-only due to the use of copy-on-write techniques and is going to be written (which will require a copy to be made).

Never migrate: On average, this page is used so infrequently that it should not be migrated to this cluster (even if it accumulates enough accesses to justify migration).

Anchor: This page will be so heavily used in this cluster that it should be anchored here and not allowed to migrate until further notice. An option to reverse this effect is also needed.

Lazy evaluation can be used to delay the effects of **always replicate** until the memory in question is actually accessed. This is done by unmapping the page in hardware and performing the operation in response to the page fault generated by the first access. This permits greater flexibility in the use of this primitive, as no additional cost is imposed for pages that are not used; similar functionality is also provided by copying pages when they are first referenced by each remote processor. Recent work [9] has indicated the utility of being able to delay making a replication or migration decision to avoid incurring large costs on the basis of transient behavior; this is an advantage of our approach over copy on reference.

These primitives can also be used to provide feedback from the management algorithms and other instrumentation over multiple runs of an application to improve its performance by adapting its memory usage to the memory structure of the machine. This feedback may reduce the effort required to restructure data to take advantage of non-uniform memory architectures.

8.8 Related Work

Competitive management of distributed shared memory is a topic at the juncture of several active areas of research. A number of researchers [11, 18, 37] have implemented distributed shared memory using messages on a network. The hardware for these implementations does not support remote accesses or access forwarding; this removes the choice of the amount of data to send in response to a request which is critical to our work. Most research projects in the area of NUMA architectures have implemented a shared memory programming model; the best known is BBN's Uniform System [49], and it is typical in that it directly exports the non-uniform memory structure to users. Our work supports automatic management mechanisms that free users from some of the details involved in managing non-uniform memory, and should make these machines easier to program. Other researchers have been implementing [9, 13] and simulating [23] memory management systems on current NUMA multiprocessors that do not provide the reference information needed by the algorithms considered in this chapter.

Some related migration results are reported by LaRowe and Ellis [33]. These results are from experiments on a Butterfly multiprocessor, and use approximations to an access count obtained by sampling and aging reference bits [23]. This sampling and aging requires a periodic software scan of all pages in the system, which introduces some additional overhead. Although this implementation is not a complete and faithful version of the migration algorithm, it should be a close enough approximation for estimation of the performance impact. They found that the competitive migration algorithm produced performance improvements among comparable algorithms, but that these performance gains were minor (no more than 5%) and were not enough to justify the additional state that the migration algorithm requires (counters per page and processor). In addition, the time taken by the periodic scans puts their implementation at a major disadvantage to migration algorithms that do not require such scans. These results suggest that this approximation to the competitive migration algorithm is not suitable to the Butterfly or similar NUMA multiprocessors. The results also suggest that the performance benefits of the actual migration algorithm may not justify the hardware investment required to achieve them, but resolving this is a topic for future work.

Chapter 9

Conclusion

The complexities of managing multiprocessor resources make it clear that a good multiprocessor operating system requires more than simple extensions to an existing uniprocessor system. Multiprocessor applications complicate the management of processor resources (scheduling) by using parallelism and concurrency (and expecting system support to achieve good performance. Non Uniform Memory Access (NUMA) multiprocessor architectures subdivide both processing and memory resources into subsets, each of which is different (e.g., each cluster of processors is close to a different pool of memory). This changes the nature of the resource management problems, because it is no longer possible to view either resource as a single large pool of interchangeable elements. Uniprocessor operating systems do not face analogous problems, and therefore lack the technology to solve them.

This thesis has explored resource management issues that arise in the design and implementation of multiprocessor operating systems. New scheduling techniques such as Handoff Scheduling and Processor Allocation have been developed to support application scheduling requirements. The processor allocation work is one of the first successful applications of policy mechanism separation to multiprocessor scheduling; this permits it to achieve an unprecedented degree of flexibility in supporting different programming models, allocation policies, and multiprocessor architectures. In the area of memory management, new competitive algorithms for managing memory in NUMA multiprocessors have been developed. In this context, 'competitive' means that their performance is within a small constant factor bound of optimal. These results are a new contribution to the field of competitive algorithm analysis. A case study using Mach addressed the major implementation issues, but an actual implementation requires hardware support that is not currently available.

9.1 Timesharing Scheduling and Timing

A new timesharing scheduler for Mach was developed as a basis for the multiprocessor scheduling research. This scheduler employs a new algorithm for computing priorities that spreads the computation over all threads (and hence all processors) in a system, avoiding the need for a periodic scan of the threads. A periodic scan of runnable threads is needed, but this scans far fewer threads because the number of runnable threads is usually a small fraction of the total number of threads. The scheduler uses an approximation technique based on two digit binary fractional expansions to replace expensive division operations with shifts and adds. Time quanta are used within the scheduler to reduce the number of context switches, and a further reduction is obtained by lengthening quanta on lightly loaded multiprocessors. This thesis also presents a closed-form solution to the expected division of processor usage among compute-bound threads at different priorities.

This work is not a detailed study of the alternatives and tradeoffs in building a timesharing scheduler, but instead demonstrates (via implementation) that a selected new design point is feasible. A comprehensive study of costs and tradeoffs (including identification of appropriate performance metrics) for timesharing scheduling is left to future work.

This thesis demonstrates that timestamp-based timing is a major improvement over statistical timing by designing and implementing a timestamp-based timing subsystem for Mach. The accuracy of statistical timing techniques is limited by the clock interrupt rate at which usage is sampled, and the results are distorted by activities synchronized to (e.g., caused by) these interrupts. Timestamp-based timing techniques avoid these distortions and can improve measurement accuracy by orders of magnitude (depending on the granularity of the timestamp source), making it possible to time processor usage down to the instruction level. Measurements of the Multimax implementation demonstrate that its overheads are small, approximately a dozen instructions per trap, interrupt, return from trap or interrupt, or context switch. These small overheads are more than justified by the improvements in accuracy. A major advantage of the Mach timing facility is that no multiprocessor (e.g., test-and-set) locks are used to protect the timers, preventing unpredictable delays to timing operations that occur in critical paths (especially interrupts). The only hardware support required is a timestamp source; a 32-bit counter running at a known clock frequency is sufficient. This improved timing facility is useful on both uniprocessors and multiprocessors.

9.2 Scheduling Support for Concurrency and Parallelism

A new classification of programming models for concurrent and parallel programs has been developed to elicit their scheduling requirements. Programming models are classified by whether they introduce concurrency at the system (virtual processor) and/or user (multiroutine) levels. The terms *virtual processor* and *multiroutine* are introduced to be precise about whether the entity is implemented by the operating system (virtual processor) or part of the application (multiroutine); the common term ‘thread’ has been used to refer to both types of entities. The resulting classification ranges from pure parallelism models that introduce no concurrency beyond the hardware parallelism to dual concurrency models that introduce concurrency at both the system and user levels. For scheduling purposes, the most important characteristic of these models is whether they introduce concurrency at the system (virtual processor) level. Programs that eschew this level of concurrency require dedicated use of processors to achieve good performance (with the exception of coroutine models). Programs that use this level of concurrency do not necessarily need dedicated processors, but can benefit from operating system support for synchronization with a virtual processor that is not running on a physical processor.

The techniques of Discouragement and Handoff Scheduling support communication and synchronization for applications that use system concurrency (more virtual processors than physical processors). Discouragement Scheduling allows running threads to supply hints to the scheduler indicating that they should not be run. Handoff Scheduling extends this by also hinting that a specific other thread should be run. These techniques are based on local information (about the thread supplying the hint and at most one other) to avoid the complexity drawbacks of basing this functionality on global information (state of the entire application). Specifying the information as hints frees the user from performing checks that the operating system can perform more efficiently (e.g., whether the target thread of a handoff is blocked). The overall design of applications supplying information (instead of the system asking for it) makes these hints optional, imposing no additional overhead on applications that do not use them. Performance results demonstrate the effectiveness of these hints in improving performance of both synchronization and communication.

Mach incorporates a new flexible approach to processor allocation. This flexibility supports applications written using different programming models, the use of different scheduling policies, and even different machine architectures(e.g., NUMA). The major contribution to this flexibility comes from isolating scheduling policy in a privileged server that is easier to replace or reconfigure than the operating system kernel. The processor allocation mechanisms are still implemented in the kernel. The kernel interface is based on the notion of a *processor set*, an independent entity to which both processors and threads can be assigned. The semantics of these assignments is that assigned threads execute only on assigned processors and vice-versa. Processor assignments are managed by the server, and thread assignments by the application. This separation of concerns frees the server from the details of application structure because the server is responsible only for assembling sets of processors in response to requests; applications are responsible for how these processors are used, and can exercise detailed control in this area if desired.

Experience with the use of Mach's processor allocation facility has been very positive. A simple server, the cpu_server, implements a basic gang scheduling policy for processes. This server bears marked similarities to a batch scheduler for a computer that does not have virtual memory. The server has been used by both parallel programming researchers and students in parallel programming courses at CMU, and has been successful in meeting their requirements for dedicated processors (primarily for performance measurement). Many extensions to this server are possible. One example has been studied and implemented under Mach by researchers from Stanford; this extension allows dynamic variation of the number of processors dedicated to an application in response to changes in system load [51]. Implementation and analysis of this and other policies for processor allocation is a productive area for future work.

9.3 NUMA Memory Management

The memory management research focuses on the development of competitive algorithms based on memory access behavior. In this context, competitive means that the performance of these algorithms is within a small constant factor of optimal, and that the performance is essentially the best that can be done without advance knowledge of future memory access behavior. Algorithms have been developed for the replication of read-only memory and the migration of writable memory on NUMA interconnection topologies corresponding to complete graphs and trees. Most actual NUMA interconnections implement a short access time to local memory and a constant longer access time to any remote memory; this corresponds to the complete graph topology. The replication algorithms are extensions of well-known results, but the migration algorithms are the first competitive solutions to the *1-server with excursions* problem. This thesis proves the competitive properties of these algorithms, and also establishes a lower bound result (3) for the 1-server with excursions problem. This result is unusual in that it has been previously proved in an earlier paper[28], but was not recognized and stated as such by the authors.

A case study of the Mach operating system demonstrates the feasibility of implementing these algorithms if the required hardware support is present. The algorithms cannot be implemented in current multiprocessor systems because they rely on reference count information that is not available from current hardware. If this information is available, the additional software changes required to implement the algorithms can be handled by Mach; existing Mach implementations of NUMA memory management [9, 13] lend support to this conclusion. Preliminary performance studies [7, 33] of these algorithms show that they can improve performance, but the demonstrated performance improvements are relatively small. This has lead the authors of one of the studies

[33] to conjecture that the performance improvements gained from their approximation to the migration algorithm are not justified by the implementation costs. More work is needed to determine the practical utility of these algorithms, and to understand the effects of the compromises that are required to approximate their behavior for implementation on the current generation of multiprocessors.

9.4 Conclusion

This thesis has demonstrated that multiprocessor operating systems require more sophisticated resource management techniques than their uniprocessor counterparts, and has made some major progress in developing such techniques. Both multiprocessor architectures and the types of applications that they are expected to support increase the complexity of the resource management problems. The work described here has concentrated on the management of processor and memory resources with the dual goals of providing better support for applications and making efficient use of hardware resources.

The processor management, or scheduling, research is concentrated on supporting requirements derived from applications. Synchronization and communication in concurrent applications are supported by Discouragement and Handoff Scheduling, which allow users to productively influence scheduling of their applications without requiring them to write their own schedulers. A new design and implementation of a Processor Allocation and Control facility for Mach supports dedication of processors to individual applications and finer control over the use of processing resources within an application, if needed. This facility achieves an unprecedented degree of flexibility in supporting different programming models, scheduling policies, and multiprocessor architectures; part of this is due to a successful application of policy mechanism separation to multiprocessor scheduling. In addition, the Mach timesharing scheduler has been redesigned to achieve a better match with multiprocessor hardware; this includes a new timestamp-based usage timing subsystem that improves timing precision and accuracy by several orders of magnitude.

In the area of memory management, this research has developed new algorithms for NUMA memory management and pioneered the use of competitive algorithms for such problems. Competitive algorithms are on-line algorithms whose performance is always within a small constant factor bound of the optimal off-line solution, even though the latter has the advantage of seeing all future events when making its decisions. The migration and replication algorithms discussed in this thesis are important contributions to the field of competitive algorithms in their own right; the migration algorithms are the first competitive solutions to the *1-server with excursions* problem, and this work also proves a tight lower-bound result for that problem. Implementation of these algorithms requires hardware support that is not present in the current generation of NUMA multiprocessors, but a feasibility study indicates that Mach should have no further difficulties in using these algorithms.

This research has made important contributions to both the principles and practice of resource management for multiprocessors and uniprocessors. The new techniques developed here should make it easier to achieve efficient and effective use of multiprocessor systems. Ongoing changes in both multiprocessor architectures (e.g., the increasing sophistication of no remote memory access, or NORMA systems) and applications (e.g., scheduling at both the system and application levels in dual concurrent programming models) continue to pose new and interesting resource management challenges. This author looks forward to the evolution of the techniques developed here and the development of new ones to meet these challenges.

Appendix A

Mach Processor Allocation Interface

This appendix describes the kernel interface for processor allocation. These primitives are used by applications and allocation servers to perform processor allocation. The primitives are divided into groups based on the objects they manipulate. Only the routines to get the host ports are traps; all other primitives are implemented by a remote procedure call message exchange with the kernel.

A.1 Host Operations

A.1.1 Get Host Port: host_self and host_priv_self

```
host_t host_self();

host_priv_t  host_priv_self();
```

Host_self and *host_priv_self* are traps that return the host port and privileged host port for the current host. The host port is used to identify and obtain information about the host. The privileged host port is used to control the physical resources on the host, in particular, it is used to obtain the processor control ports that allow processors to be assigned. *Host_priv_self* currently uses Unix authentication; it will return the null port if the invoker does not have the privileges of the Unix super-user.

Both of these traps are slated to vanish in future versions of Mach. The *host_self* trap will be replaced by an extension of *task_get_special_port* to return the host port. This allows interposition on the host port on a per-task basis (e.g., for debugging applications and servers that use it). *Host_priv_self* will be replaced by a mechanism that inserts the privileged host port in the port space of the first task to be run on the host as part of the boot sequence; that task is then responsible for implementing the appropriate authentication functionality to restrict access to the port.

A.1.2 Get Processor Control Ports: host_processors

```
kern_return_t
host_processors(host_priv, processor_list, processor_count)
        host_priv_t       host_priv;
        processor_array_t *processor_list; /* out, pointer to list */
        int               *processor_count; /* out */
```

Host_processors obtains the control ports for all processors on a host. The ports are returned in an array, but the position of each port in the array may not correspond to any architectural feature of the machine (e.g., slot numbers). *Processor_info* should be used to obtain information about the processors represented by the ports. The ports are returned in newly allocated memory; the caller may wish to deallocate it when the data is no longer needed. The privileged host port from *host_priv_self* must be used.

A.1.3 Obtain Host Information: host_info

```
kern_return_t host_info(host, flavor, info, count)
        host_t              host;
        int                 flavor;
        host_info_t         info;               /* in and out */
        unsigned int        *count;             /* in and out */
```

Host_info returns information about the host. Either the host port or the privileged host port may be used. Three flavors of information are currently supported: basic information, scheduling information, and active processor slots.

Basic information is defined by the following structure:

```
   struct host_basic_info {
        int             max_cpus;       /* maximum possible cpus for
                                           this kernel */
        int             avail_cpus;     /* number of cpus available */
        vm_size_t       memory_size;    /* size of memory in bytes */
        cpu_type_t      cpu_type;       /* cpu type */
        cpu_subtype_t   cpu_subtype;    /* cpu subtype */
   };
```

The definitions for the cpu type and subtype are contained in the file `<mach/machine.h>`.

Scheduling information is defined by the following structure:

```
struct host_sched_info {
        int             min_timeout;    /* minimum timeout */
        int             min_quantum;    /* minimum quantum */
};
```

Milliseconds are used as the units for these quantities to correspond to the timeout interface for IPC.

The active processor slots flavor returns a list of the slot numbers of the processors that are currently active (on line) for this host.

The introduction of multiple flavors represents a change to the calling sequence of the *host_info* call in previous kernels, but a new call number has been used so that old programs will continue to work. This call can be extended with machine-dependent flavors to return important machine-dependent configuration information.

A.1.4 Obtain Kernel Version: host_kernel_version

```
kern_return_t host_kernel_version(host,version)
        host_t                  host;
        kernel_version_t        *version;
```

Host_kernel_version returns the version string for the kernel executing on the given host. Either the host port or the privileged host port may be used. Experience indicates that this is the only reliable way to determine what's actually running; version numbers are inadequate.

A.1.5 Obtain List of Processor Sets: host_processor_sets

```
kern_return_t
host_processor_sets(host, processor_set_name_list, processor_set_count)
    host_t                host;
    processor_set_name_array_t *processor_set_list; /* out, ptr to array */
    int                   *processor_set_count;        /* out */
```

Host_processor_sets returns a list of the processor sets that currently exist on the specified host. The sets are identified by their name ports. Either the host port or the privileged host port may be used. This list is returned in an array that is created as a result of the call. The caller may wish to deallocate it when the data is no longer needed.

A.2 Processor Operations

A.2.1 Obtain Information: processor_info

```
kern_return_t
processor_info(processor, flavor, host, processor_info, count)
        processor_t        processor;
        int                flavor;
        host_t             *host;            /* out */
        processor_info_t   processor_info;   /* in and out */
        int                *count;           /* in and out */
```

Processor_info returns information about a processor, including the host to which it belongs. This is the ordinary host port, not the privileged host port. Only one machine-independent flavor of information is defined:

```
struct processor_basic_info {
        cpu_type_t      cpu_type;       /* cpu type */
        cpu_subtype_t   cpu_subtype;    /* cpu subtype */
        boolean_t       running;        /* is processor running? */
        int             slot_num;       /* slot number */
        boolean_t       is_master;      /* is processor the master? */
};
```

Machine-dependent flavors can be defined to return additional machine-dependent configuration information.

A.2.2 Start Processor: processor_start

```
kern_return_t processor_start(processor)
        processor_t        processor;
```

Processor_start starts the specified processor. Support for this call is machine-dependent; on machines that do not support it, nothing happens.

A.2.3 Stop Processor: processor_exit

```
kern_return_t processor_exit(processor)
        processor_t         processor;
```

Processor_exit exits a processor from the system. Support for this call is machine-dependent, as is the ability to restart an exited processor.

A.2.4 Miscellaneous: processor_control

```
kern_return_t processor_control(processor, command, count)
        processor_t         processor;
        int                 *command;        /* array of ints */
        int                 count;                /* length of array */
```

Processor_control issues the specified command to the specified processor. Interpretation of this command is machine-dependent. This routine is useful for performing console functions on multiprocessors that have one logical console per processor rather than a single system console.

A.3 Processor Set Operations

A.3.1 Creation: processor_set_create

```
kern_return_t processor_set_create(host, new_set, new_name)
        host_t                    host;
        processor_set_t           *new_set;
        processor_set_name_t      *new_name;
```

Processor_set_create creates a new processor set on the specified host and returns the two ports associated with it. The port returned in *new_set* is the actual port representing the set. It is used to perform operations such as assigning processors, tasks, or threads. The port returned in *new_name* identifies the set, and is used to obtain information about the set. Usually the ordinary host port will be used, but the privileged host port is also accepted.

A.3.2 Destruction: processor_set_destroy

```
kern_return_t processor_set_destroy(processor_set)
        processor_set_t                 processor_set;
```

Processor_set_destroy destroys the specified processor set. Any assigned processors, tasks, or threads are reassigned to the default set. The object port for the processor set is required (not the name port). The default processor set cannot be destroyed.

A.3.3 Identify Default Set: processor_set_default

```
kern_return_t processor_set_default(host, default_set)
        host_t                host;
        processor_set_t       *default_set;
```

The default processor set is used by all threads, tasks, and processors that are not explicitly assigned to other sets. *Processor_set_default* returns a port that can be used to obtain information about this set (e.g., how many threads are assigned to it). This port cannot be used to perform operations on that set.

A.3.4 Obtain Information: processor_set_info

```
kern_return_t
processor_set_info(processor_set, flavor, host, info, count)
        processor_set_t       processor_set;
        int                   flavor;
        host_t                *host;
        processor_set_info_t  info;       /* in and out */
        int                   *count;     /* in and out */
```

Processor_set_info returns the requested information (specified by the *flavor* argument for a processor set. Either the processor set's name port or object port may be used for this call. The *info* structure is an array of integers that is supplied by the caller and filled with the requested information. *count* specifies the size of the array and returns the size of the provided information.

Basic information is defined by the following structure:

```
struct processor_set_basic_info {
        int                processor_count; /* number of processors */
        int                task_count;      /* number of tasks */
        int                thread_count;    /* number of threads */
        int                load_average;    /* scaled load average */
        int                mach_factor;     /* scaled mach factor */
};
```

The load average is the average number of runnable threads. The mach factor is the average number of idle processors. Both are scaled by the constant `LOAD_FACTOR` (e.g., if the actual value is 1, the integer value returned is `LOAD_FACTOR`).

Additional scheduling information is defined by the following structure:

```
struct processor_set_sched_info {
        int             policies;       /* allowed policies */
        int             max_priority;   /* max priority  */
};
```

The *policies* field is a bit mask of the policies allowed by the processor set.

A.3.5 Get List of Tasks: processor_set_tasks

```
kern_return_t processor_set_tasks(processor_set, task_list, task_count)
        processor_set_t   processor_set;
        task_array_t      *task_list;    /* out, ptr to array */
        int               *task_count;   /* out */
```

Processor_set_tasks returns a list of tasks currently assigned to the specified processor set. Tasks are identified by their kernel port. The array of tasks is created as a result of this call, and the caller may wish to deallocate it when the information is no longer needed. The processor set's object port is required for this call.

A.3.6 Get List of Threads: processor_set_threads

```
kern_return_t
processor_set_threads(processor_set, thread_list, thread_count)
        processor_set_t  processor_set;
        thread_array_t   *thread_list;      /* out, ptr to array */
        int              *thread_count;     /* out */
```

Processor_set_threads returns a list of threads currently assigned to the specified processor set. Threads are identified by their kernel port. The array of threads is created as a result of this call, and the caller may wish to deallocate it when the information is no longer needed. The processor set's object port is required for this call.

A.3.7 Get Control Port: host_processor_set_priv

```
kern_return_t host_processor_set_priv(host_priv, set_name, set)
        host_priv_t           host_priv;
        processor_set_name_t  set_name;
        processor_set_t       *set;
```

Host_processor_set_priv allows a privileged application (one with rights to the privileged host port) to obtain the control port for a processor set from its name port. This is primarily intended for recovery from situations in which an application has been suspended because all of its processors have been reassigned. Obtaining the control port allows the processor set to be destroyed, or a new allocation of processors to be requested (from the server).

A.4 Execution Control Operations

A.4.1 Processor Assignment: processor_assign

```
kern_return_t processor_assign(processor, processor_set, wait)
        processor_t      processor;
        processor_set_t  processor_set;
        boolean_t        wait;
```

Processor_assign assigns a processor to a processor set. After this assignment, the processor only executes threads assigned to the set. Any previous assignment is nullified. A *wait* argument of TRUE causes this call to synchronize with the specified processor ceasing to execute its current thread if that thread is not in the specified set. Using a wait argument of FALSE permits application-kernel parallelism to be exploited because the kernel uses a separate internal thread (or threads) to perform processor assignment. The master processor cannot be reassigned. It remains assigned to the default processor set to ensure that internal kernel threads and important user daemons can be run. Redirection of device interrupts away from processors assigned to sets other than the default set is machine-dependent. All processors take clock interrupts at all times.

A.4.2 Obtain Processor Assignment: processor_get_assignment

```
kern_return_t processor_get_assignment(processor, assigned_set_name)
        processor_t          processor;
        processor_set_name_t *assigned_set_name;  /* out */
```

Processor_get_assignment returns the name port of the processor set to which the processor is currently assigned.

A.4.3 Thread Assignment: thread_assign

```
kern_return_t thread_assign(thread, processor_set)
        thread_t        thread;
        processor_set_t processor_set;
```

Thread_assign assigns a thread to a processor set. After assignment, the thread only executes on processors assigned to that set. Any previous assignment is nullified. This call synchronizes with the specified thread switching away from its current processor if that processor is not in the specified set. Unix system call compatibility code in some kernels may temporarily force threads to execute on the master processor.

A.4.4 Default Thread Assignment: thread_assign_default

```
kern_return_t thread_assign_default(thread)
        thread_t        thread;
```

Thread_assign_default is a version of *thread_assign* that assigns the thread to the default processor set on its host. This call is necessary because the control port for the default processor set is not available to unprivileged applications.

A.4.5 Obtain Thread Assignment: thread_get_assignment

```
kern_return_t thread_get_assignment(thread, processor_set)
        thread_t                thread;
        processor_set_name_t    *processor_set;  /* out */
```

Thread_get_assignment returns the name port of the processor set to which the thread is currently assigned.

A.4.6 Task Assignment: task_assign

```
kern_return_t task_assign(task, processor_set, assign_threads)
        task_t           task;
        processor_set_t  processor_set;
        boolean_t        assign_threads;
```

Task_assign assigns a task to a processor set. This assignment is for the purposes of obtaining the initial assignment for newly created threads. Existing threads are also assigned if *assign_threads* is TRUE. These assignments are as if *thread_assign* was called for each thread.

A.4.7 Default Task Assignment: task_assign_default

```
kern_return_t task_assign_default(task, assign_threads)
        task_t      task;
        boolean_t   assign_threads;
```

Task_assign_default is a version of *task_assign* that assigns the task to the default processor set on its host. This call is necessary because the control port for the default processor set is not available to unprivileged applications.

A.4.8 Obtain Task Assignment: task_get_assignment

```
kern_return_t task_get_assignment(task, processor_set)
        task_t                task;
        processor_set_name_t  *processor_set;  /* out */
```

Task_get_assignment returns the name port of the processor set to which the task is currently assigned.

Appendix B

Mach Priority and Policy Interface

This appendix describes the kernel interface for scheduling policies and priorities. Only timesharing and fixed priority policies exist, but the interface is extensible to allow the definition and use of other policies in the future. The timesharing policy is for the usual timeshared use of a machine. The fixed priority policy is intended to support soft real time applications. All processor set arguments in this appendix denote the control port for the processor set.

B.1 Priority Operations

B.1.1 Set Thread Priority: thread_priority

```
kern_return_t thread_priority(thread, priority, set_max)
        thread_t   thread;
        int        priority;
        boolean_t  set_max
```

Thread_priority sets the priority of a thread, and also its maximum priority, if *set_max* is TRUE. This priority is the base priority for the thread; the actual priority used for scheduling is computed by the thread's scheduling policy. This call will fail if an attempt is made to raise a thread's priority above its current maximum priority.

B.1.2 Change Thread Max Priority: thread_max_priority

```
kern_return_t thread_max_priority(thread, processor_set, priority)
        thread_t        thread;
        processor_set_t processor_set;
        int             priority;
```

thread_max_priority changes the maximum priority of a thread. It requires presentation of the control port for the processor set to which the thread is currently assigned to establish the caller's right to raise the maximum priority. The maximum priority can be lowered by the *set_max* option to *thread_priority* without the need to present this port. If a thread's priority is higher than the new maximum priority set by this call, the thread's priority is lowered to the new maximum.

B.1.3 Set Task Priority: task_priority

```
kern_return_t task_priority(task, priority, change_threads)
        task_t      task;
        int         priority;
        boolean_t   change_threads;
```

The priority of a task is used only for creation of new threads; a new thread's priority is set to the enclosing task's priority. *task_priority* changes this task priority. It also sets the priorities of all threads in the task to this new priority if *change_threads* is TRUE. Otherwise, existing threads are not affected. If this priority change violates the maximum priority of some threads, as many threads as possible will be changed and an error code will be returned.

B.1.4 Set Max Priority for Processor Set: processor_set_max_priority

```
kern_return_t
processor_set_max_priority(processor_set, priority, change_threads)
        processor_set_t           processor_set;
        int                       priority;
        boolean_t                 change_threads;
```

The priority of a processor set is used only for newly created threads (thread's maximum priority is set to processor set's) and the assignment of threads to the set (thread's maximum priority is reduced if it exceeds the set's maximum priority, thread's priority is similarly reduced). *processor_set_max_priority* changes this priority. It also sets the maximum priority of all threads assigned to the processor set to this new priority if *change_threads* is TRUE. If this maximum priority is less than the priorities of any of these threads, their priorities will also be set to this new value.

B.2 Scheduling Policy Operations

B.2.1 Set Thread Policy: thread_policy

```
kern_return_t thread_policy(thread, policy, data)
        thread_t        thread;
        int             policy;
        int             data;
```

thread_policy changes the scheduling policy for a thread. *data* is policy-dependent scheduling information. There are currently two supported policies: *POLICY_TIMESHARE* and *POLICY_FIXEDPRI*. these symbols are defined in the file ¡mach/policy.h¿, which is included by ¡mach.h¿. *data* is meaningless for timesharing, but is the quantum to be used (in milliseconds) for the fixed priority policy. To be meaningful, this quantum must be a multiple of the basic system quantum (*min_quantum*) which can be obtained from *host_info*. The system will always round up to the next multiple of the quantum. Processor sets may restrict the allowed policies, so this call will fail if the processor set to which the thread is currently assigned does not permit the specified policy.

B.2.2 Enable Policy: processor_set_policy_enable

```
kern_return_t processor_set_policy_enable(processor_set, policy)
        processor_set_t processor_set;
        int             policy;
```

Allow use of the specified policy by threads assigned to the processor set.

B.2.3 Disable Policy: processor_set_policy_disable

```
kern_return_t
processor_set_policy_disable(processor_set, policy, change_threads)
        thread_t        thread;
        processor_set_t processor_set;
        int             change_threads;
```

Disallow use of the specified policy for the specified processor set. *change_threads* indicates whether this call should affect existing threads. If it is TRUE, any existing thread with the specified policy has its policy reset to timesharing. If it is FALSE, existing threads with the policy are grandfathered; they keep their policy, but threads subsequently created in the set or assigned to it with this policy will have it reset to timesharing. The timesharing policy may not be disabled.

Appendix C

The cpu_server Interfaces

This appendix documents the interfaces to the Mach cpu_server. Both the low level remote procedure call interface and the library interfaces built on top of it are included.

C.1 Server RPC Interface

This section describes the basic remote procedure call (RPC) interface used to access the server. This interface is based on the concept of a *request* object that is created and manipulated by these primitives.

C.1.1 Create a Request: cpu_request_create

```
mach_error_t
cpu_request_create(server, total_processors, run_time, delay, request)
        port_t          server;
        int             total_processors;
        int             run_time;
        int             *delay;         /* out */
        cpu_request_t   *request;       /* out */
```

Cpu_request_create creates a cpu_server request object for a total of *total_processors* processors and a maximum of *run_time* seconds. The newly created request object and a current delay estimate are returned. The request is created in the inactive state. The initial server port is obtained by looking up the name 'cpu_server' with the local nameservice.

C.1.2 Add to a Request: cpu_request_add

```
mach_error_t
cpu_request_add(request, processor_set, processors, processors_left)
        cpu_request_t    request;
        processor_set_t  processor_set;
        int              processors;
        int              *processors_left;   /* out */
```

Cpu_request_add adds a request for processors for the indicated processor set to the specified request. This number of processors is deducted from the total number of processors specified when the request was created, and the number remaining to be associated with a processor set is returned. The request must be in the inactive state for this call.

C.1.3 Request Notifications: cpu_request_set_notify

```
mach_error_t
cpu_request_set_notify(request, notify_port)
        cpu_request_t   request;
        port_t          notify_port;
```

Cpu_request_set_notify asks the server to send a notification message to *notify_port* after allocating processors to the request, and one second before removing them. Applications that receive these messages can use them to make sure that execution only takes place while all of the processors are allocated; the end message can be used to initiate a barrier synchronization to stop cleanly, and the start message can be used to exit the barrier. The request must be in the inactive state for this call.

C.1.4 Activate Request: cpu_request_activate

```
mach_error_t
cpu_request_activate(request, options, total_time, delay)
        cpu_request_t  request;
        int            options;
        int            total_time;
        int            *delay;  /* out */
```

Cpu_request_activate activates a request (i.e., requests the server to allocate processors to it). A maximum delay until the processors will be allocated is returned. No further *cpu_request_add* or *cpu_request_set_notify* operations are permitted on the request. The following options are supported:

- *CPU_OPTION_DESTROY* - Destroy the processor sets when the request is completed. This is intended to support naive users by preventing a program that overruns its request from going into suspended animation; destroying its processor sets forces the program back into the default processor set where it will continue to run.

- *CPU_OPTION_REPEAT* - Repeat the request for *total_time*. The time for each instance of the request is specified by the run_time argument of the *cpu_request_create* operation that created the request. *total_time* is ignored if this option is not selected.

The symbols for these options are defined in the file ¡servers/cpu_server.h¿.

C.1.5 Activate Request for Task: cpu_request_activate_task

```
mach_error_t
cpu_request_activate_task(request, options, total_time, task, delay)
        cpu_request_t  request;
        int            options;
        int            total_time;
        task_t         task;
        int            *delay;  /* out */
```

This is a version of *cpu_request_activate* which informs the server that the application using the processor sets is a task, and identifies the task. This allows the server to optimize assignment processors by suspending the task because processor assignment is faster if the processor is idle.

C.1.6 Terminate a Request: cpu_request_destroy

```
mach_error_t cpu_request_destroy(request)
        cpu_request_t        request;
```

Cpu_request_destroy destroys the specified request, freeing any processors that were allocated to it. If notifications were requested for this request, an end notification will be sent and the freeing of the processors will be delayed by 1 second.

C.1.7 Get Request Information: cpu_request_status

```
mach_error_t
cpu_request_status(request, reserved, assigned, active, options, time)
        cpu_request_t request;
        int           *reserved;  /* out */
        int           *assigned;  /* out */
        boolean_t     *active;    /* out */
        int           *options;   /* out */
        int           *time;      /* out */
```

Cpu_request_status obtains information about an allocation request. *active* indicates whether the request has been activated. For an inactive request, *reserved* is the number of processors reserved when the request was created, and *assigned* is the number of processors used by *cpu_request_add* calls. For an active request, *reserved* is the number of processors needed to satisfy the request, and *assigned* is the number currently assigned; because processor assignment is done by a single thread, this number may have values between zero and *reserved*. *options* indicates the options currently associated with the request; notification is an option, even though it is requested in a separate call. *time* is the server's estimate of the amount of time until the request can be satisfied if the request is not active or has no processors assigned. It is the time until the processors will be deactivated (or the end notification sent, if requested) for an active request that has processors assigned.

C.1.8 Get Server Information: cpu_server_info

```
mach_error_t
cpu_server_info(server, max_time, max_total_time, max_processors,
    delay)
        cpu_request_t   server;
        int             *max_time;          /* out */
        int             *max_total_time;    /* out */
        int             *max_processors;    /* out */
        int             *delay;             /* out */
```

Cpu_server_info obtains information about the server. *max_time* is the maximum time in seconds for a single request or single segment of a repeating request. *max_total_time* is the maximum total time for a repeating request (one that uses the Repeat option). *max_processors* is the maximum number of processors available to any request, and *delay* is the server's estimate of the maximum delay until a new request can be satisfied. The *cpuinfo* program uses this call to query the server.

C.2 Library Interfaces

The server interface and the kernel interface for processor allocation will be used directly by programs that require explicit control over which threads are executing on which processors at which time. For applications with less stringent processor allocation requirements, simple library interfaces which hide all of the internal details of interaction with the kernel and server may be appropriate. Four interfaces have been developed: *allocate*, *task*, *hook*, and *task-hook*.

C.2.1 Allocate Interface

The *allocate* interface supports a single allocation of a pool of processors. It exports the following two calls:

```
allocate_processors(num_processors, num_seconds, interactive)
        int          num_processors;
        int          num_seconds;
        boolean_t    interactive;

deallocate_processors()
```

Allocate_processors allocates *num_processors* processors for *num_seconds* seconds. If the time is larger than the server's maximum slice time, then a repeating request is automatically submitted. If *interactive* is TRUE, errors generate printfs, and the user is asked whether the server's maximum delay is acceptable; no allocation is performed if the answer is no. *allocate_processors* does not return until the allocation of processors has started; it performs a *task_assign* internally so that the initial thread and all threads and tasks subsequently created share the allocated processors. If a program overruns its time allocation, it will continue to run, but without dedicated processors.

Deallocate_processors frees the processors allocated by *allocate_processors*. It must be called by a thread in the same task that did the allocation.

C.2.2 Task Interface

The *task* interface is identical to the allocate interface, but is restricted to applications consisting of a single task so that the server can exploit efficiencies available in this case (suspending the task before removing processors). The task interface exports the following two calls:

```
task_allocate_processors(num_processors, num_seconds, interactive)
        int         num_processors;
        int         num_seconds;
        boolean_t   interactive;

task_deallocate_processors()
```

C.2.3 Hook Interface

The *hook* interface supports allocation of a pool of processors, with user scheduling hooks. It exports the following two calls:

```
allocate_processors_with_hooks(num_processors, num_seconds, start_hook,
    end_hook, interactive)
        int       num_processors;
        int       num_seconds;
        int       (*start_hook)();
        int       (*end_hook)();
        boolean_t interactive;

deallocate_processors_with_hooks()
```

Allocate_processors_with_hooks allocates the specified number of processors for the specified number of seconds. If the number of seconds is greater than the server's maximum slice time, a repeating request is automatically submitted. *start_hook* is called each time after the processors are allocated. Similarly, *end_hook* is called approximately 1 second before any processor deallocation. *interactive* functions identically to the previous interfaces. A thread must be dedicated to the *allocate_processors_with_hooks* call; i.e., the calling thread does not return until the allocation request is finished or terminated. This dedicated thread is assigned to the allocated processors (this can be reversed by performing a *thread_assign_default* or *thread_assign* as part of the first invocation of *start_hook*). All threads within its task and subsequently created tasks are also assigned to the allocated processors (c.f. *task_assign*). Both *start_hook* and *end_hook* must return for the interface to function correctly. In particular, the interface will break if *end_hook* does not return before the processors are deallocated.

Deallocate_processors_with_hooks frees the processors allocated by *allocate_processors_with_hooks*. *end_hook* will not be invoked as a result of this call, but users should be aware of the race between this deallocation and the server deallocating because a time-slice has ended.

C.2.4 Task-hook Interface

Finally, there is the *task-hook* interface, which combines the functionality of the hook interface with the server optimization of the task interface; the calls in this interface are the hook interface calls with *task_* prefixed. Users should note that the interfaces are independent; processors must be deallocated with the deallocate call from the same interface that was used to allocate them.

Appendix D

Thread_switch Interface

This appendix documents the kernel interface to *thread_switch* and related calls that perform Discouragement and Handoff Scheduling.

```
kern_return_t thread_switch(new_thread, option, time)
        thread_t   new_thread;
        int        option;
        int        time;
```

thread_switch provides low-level access to the scheduler's context switching code. *new_thread* is a hint that implements handoff scheduling. The operating system will attempt to switch directly to the new thread (bypassing the normal logic that selects the next thread to run), if possible. Since this is a hint, it may be incorrect; it is ignored if it doesn't specify a thread on the same host as the current thread or if that thread can't be switched to (not runnable or already running on another processor). In this case, the normal logic to select the next thread to run is used; the current thread may continue running if there is no other appropriate thread to run.

Options for *option* are defined in ¡mach/thread_switch.h¿. Three options are recognized:

1. `SWITCH_OPTION_NONE` – No options, *time* is ignored.

2. `SWITCH_OPTION_WAIT` – The thread is blocked for the specified *time*. This can be aborted by *thread_abort*.

3. `SWITCH_OPTION_DEPRESS` – The thread's priority is depressed to the lowest possible value for *time*. The priority is reset by the scheduler running the thread again, expiration of the timeout, or a *thread-abort* operation on this thread.

The priority depression is independent of operations that change the thread's priority (e.g., *thread_priority* will not abort the depression). The minimum time and units of time can be obtained as the *min_timeout* value from *host_info*. *thread_switch* is an optimized trap that operates on the thread that invokes it.

Users should beware of calling *thread_switch* with an invalid hint (e.g., `THREAD_NULL`) and no option. Because the time-sharing scheduler varies the priority of threads based on usage, this may result in a waste of cpu time if the thread that must be run is of lower priority. The use of the DEPRESS option in this situation is highly recommended.

thread_switch ignores policies. Users relying on the preemption semantics of a fixed time policy should be aware that *thread_switch* ignores these semantics; it will run the specified *new_thread* independent of its priority and the priority of any other threads that could be run instead.

Appendix E

Timer Interface

This appendix describes the interface to the module that implements the accurate timers within the kernel. It is only available within the kernel; applications can obtain usage times from the timers via the *task_info* and *thread_info* kernel operations.

Each thread has both a user-mode and a system-mode timer. In addition, there is at least one set of kernel overhead timers (a set of kernel timers consists of one per processor). In addition to the timer data structures, this module also maintains a data structure that indicates the currently running timer for each processor in the system (there is always exactly one running timer for each processor at any time).

E.1 Initialization Routines

E.1.1 Master Initialization: init_timers

```
init_timers()
```

Init_timers is called by the master processor (first processor) during the boot sequence. It initializes all of the overhead timers and starts a kernel overhead timer for the master processor.

E.1.2 Slave Initialization: start_timer

```
start_timer(timer)
        timer_t         timer;
```

Start_timer is called once by each processor in the boot sequence to start its kernel overhead timer. The master processor calls it from *init_timers*, and slaves call it directly from their startup code.

E.1.3 Termination

There is currently no termination routine to stop a timer for a processor that exits the system.

E.2 Trap and Context Switch Routines

These routines are called by the trap and context switch code to switch the running timer. They are all versions of the **Switch** operation; stop one timer and start another using the same timestamp. Implementors will probably want to use inline assembly language for these routines to gain speed.

E.2.1 Trap Entry: time_trap_uentry

```
time_trap_uentry(ts)
        unsigned        ts;
```

Time_trap_uentry does trap entry timing. The caller must lock out interrupts and take a timestamp. *ts* is a timestamp taken after interrupts are locked out. This routine stops the current timer on this processor and starts the system timer for the current thread. The current timer (which must be the user timer for the current thread) is checked for overflow, and *timer_normalize* is called, if needed. This routine must only be called if the trap was from user mode.

E.2.2 Trap Exit: time_trap_uexit

```
time_trap_uexit(ts)
        unsigned        ts;
```

Time_trap_uexit does trap exit timing. The caller must lock out interrupts and take a timestamp. *ts* is a timestamp taken after interrupts are locked out. This routine stops the current timer on this processor and starts the user timer for the current thread. The current timer (which must be the system timer for the current thread) is checked for overflow, and *timer_normalize* is called, if needed. This routine must only be called if the trap return is to user mode.

E.2.3 Interrupt Entry: time_int_entry

```
timer_t time_int_entry(ts,new_timer)
        unsigned ts;
        timer_t  new_timer;
```

Time_int_entry does interrupt entry timing. The caller must lock out interrupts and take a timestamp. *ts* is a timestamp taken after interrupts are locked out. *new_timer* is the new timer to switch to. This routine stops the current timer and switches to the new timer (which must be an overhead timer for the current processor). This routine returns the timer that was running when it was called; this timer (pointer to data structure) MUST be pushed onto the stack by the caller, or otherwise saved for *time_int_exit*.

E.2.4 Interrupt Exit: time_int_exit

```
time_int_exit(ts, old_timer)
        unsigned ts;
        timer_t  old_timer;
```

Time_int_exit does interrupt exit timing. The caller must lock out interrupts and take a timestamp. *ts* is a timestamp taken after interrupts are locked out. *old_timer* is the timer value pushed onto the stack or otherwise saved after *time_int_entry* returned it. This routine stops the current timer and switches to *old_timer*. Both the current timer and *old_timer* are checked for overflow, and *timer_normalize* is called, if needed.

E.2.5 Context Switch: timer_switch

```
timer_switch(new_timer)
        timer_t new_timer;
```

Timer_switch switches to a new timer. It uses the machine dependent routine or macro `get_timestamp()` to obtain a timestamp which is used to stop the current timer and start *new_timer*. The current timer is checked for overflow and *timer_normalize* is called if needed. The caller must lock out interrupts.

E.3 Basic Timer Operations

This section describes the basic routines that normalize a timer and read its value. These are the routines that incorporate the synchronization derived from Lamport's clock algorithm[31].

E.3.1 Normalize: timer_normalize

```
timer_normalize(timer)
        timer_t        timer;
```

Timer_normalize normalizes a timer value. It is called from the trap, interrupt, and context switch routines if the most significant bit in the low order accumulated time word is set in a timer.

E.3.2 Read: timer_grab

```
timer_grab(timer, save)
        timer_t         timer;
        timer_save_t    save;
```

Timer_grab is the basic **Read** routine for timers. It returns a save value, which is a copy of the low and high order accumulated time fields of the timer that is guaranteed to be consistent.

E.3.3 Read Routines

These routines read accumulated time values from timers. A `time_value_t` contains fields for seconds and microseconds.

E.3.4 Read a Timer: timer_read

```
timer_read(timer, tv)
        timer_t      timer;
        time_value_t *tv;
```

Timer_read reads the value of a timer and returns it normalized in second and microsecond units.

E.3.5 Read both Thread Timers: thread_read_times

```
thread_read_times(thread, user_time_p, system_time_p)
        thread_t        thread;
        time_value_t    *user_time_p;
        time_value_t    *system_time_p;
```

thread_read_times reads the values of both the user and system timers for a specified thread and returns them normalized in second and microsecond units.

E.4 Scheduler Interface

This section describes the routines used by the scheduler to obtain usage timing information.

E.4.1 Delta: timer_delta

```
unsigned timer_delta(timer, save)
        timer_t         timer;
        timer_save_t    save;
```

Timer_delta takes the difference of a saved timer value and the current one, and updates the saved value to current. The difference is returned as a function value. This routine handles the general case in which the high order save value and the high order timer value may not match.

E.4.2 Fast Delta: TIMER_DELTA

```
TIMER_DELTA(timer, save, result)
        timer_t         timer;
        timer_save_t    save;
        unsigned        result;
```

TIMER_DELTA is a macro form of *timer_delta* that optimizes the common case of the high save value and high timer value matching. If these values don't match, *timer_delta* is called to do this operation the slow way.

E.4.3 Delta for Scheduler: thread_timer_delta

```
thread_timer_delta(thread)
        thread_t        thread;
```

Thread_timer_delta updates the scheduler's timer information for a thread. It calls *TIMER_DELTA* on both the system and user timers for the thread, and updates the corresponding save values (all in the thread's data structure). The delta values are used to keep running counts of cpu usage (unweighted) and scheduler usage (weighted by overload factor). The scheduler uses these values to compute percent cpu usage and priority for a timesharing thread, respectively.

E.5 Timestamps and Units

This section describes the interface between the timer code and the source of timestamps. These interfaces are normally defined in the machine-dependent header file machine/timer.h.

E.5.1 Obtain Timestamp: get_timestamp

The `get_timestamp()` routine or macro must return a 32-bit timestamp from the timestamp source.

E.5.2 Describe Timestamps

`TIMER_MAX` defines the value at which the timestamp source rolls over. If the difference between two timestamps is negative, this value is added to correct for the (assumed) rollover. No such correction is needed for timers that roll over at their full 32-bit value, so `TIMER_MAX` is not defined for them.

`TIMER_RATE` defines the rate of the timestamp source in counts per second (hertz). It is used by the scheduler to calculate percent cpu usage.

`TIMER_HIGH_UNIT` defines the number of timestamp units (`TIMER_RATE` per second) that constitute one unit for the high word of the timer. Setting this to `TIMER_RATE` causes the high word to use units of seconds.

E.5.3 Unit Adjustment

`TIMER_ADJUST()` adjusts for timestamp units for read operations. It takes a `timer_save_t` as an argument and converts it to use low units of microseconds and high units of seconds. This is a null operation and not defined if the timer low and high units are already microseconds and seconds. The timer low unit always matches the unit of the timestamp source.

`PRI_SHIFT` and `PRI_SHIFT_2` adjust for timestamp units for scheduler delta operations. They define a two shift and add or subtract optimization as described in Section 2.2.3, except that `PRI_SHIFT_2` is ignored if it is not defined. For timestamp units of microseconds, `PRI_SHIFT` is 16 for the 0–127 priority range, and 18 for the 0–31 priority range. `PRI_SHIFT_2` is not defined in these cases. These shift values can be used to calculate the appropriate shifts for other timestamp units.

Bibliography

[1] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[2] A. J. Bernstein and J. C. Sharp. A Policy-Driven Scheduler for a Time-Sharing System. *Communications of the ACM*, 14(2):74–78, February 1971.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[4] Andrew D. Birrell. An Introduction to Programming with Threads. Technical Report 35, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, 1989.

[5] Roberto Bisiani, Andreas Nowatzyk, and Mosur Ravishankar. Coherent Shared Memory on a Message Passing Machine. Technical Report CMU-CS-88-204, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1988.

[6] Roberto Bisiani and Mosur Ravishankar. PLUS: A Distributed Shared Memory System. In *Conference Proceedings: The 17th International Symposium on Computer Architecture*, pages 115–124, May 1990.

[7] David Black, Anoop Gupta, and Wolf-Dietrich Weber. Competitive Management of Distributed Shared Memory. In *Proceedings, Spring Compcon '89*, pages 184–190, San Francisco, CA, February 1989. IEEE Computer Society.

[8] William Bolosky. Personal Communication, September 1988. Performance results from the Mach implementation on the ACE workstation.

[9] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[10] Ray Bryant. The RP3 Parallel Computing Environment. In *Proceedings of USENIX 1988 Supercomputer Workshop*, pages 69–85, 1988.

[11] David Cheriton. Unified Management of Memory and File Caching Using the V Virtual Memory System. Technical Report STAN-CS-88-1192, Computer Science Dept., Stanford University, Stanford, CA, 1988.

[12] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, June 1988.

[13] Alan L. Cox and Robert J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[14] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance Mmeasurements on a 128-node Butterfly Parallel Processor. In *Intl. Conf. on Parallel Processing*, pages 531–540, 1985.

[15] Raymond B. Essick. An Event-based Fair-Share Scheduler. In *1990 USENIX Winter Conference Proceedings*, pages 147–161, January 1990.

[16] Dror G. Feitelson and Larry Rudolph. Distributed Hierarchical Control for Parallel Processing. *COMPUTER*, 23(5):65–77, May 1990.

[17] Amos Fiat, Richard Karp, Michael Luby, Lyle McGeoch, Daniel Sleator, and Neal Young. Competitive Paging Algorithms. Technical Report CMU-CS-86-164, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1986.

[18] Alessandro Forin, Joseph Barrera, and Richard Sanzi. The Shared Memory Server. In *Winter USENIX Conference Proceedings*, pages 229–244, San Diego, CA, February 1989. USENIX Association.

[19] Armando Garcia, David Foster, and Richard Freitas. The Advanced Computing Environment Multiprocessor Workstation. Research Report RC14491, IBM T. J. Watson Research Center, Hawthorne, NY, 1988.

[20] Edward F. Gehringer, Daniel P. Siewiorek, and Zary Segall. *Parallel Processing: The Cm* Experience*. Digital Press, Maynard, MA, 1987.

[21] Alan Gottlieb. The NYU Ultracomputer—Designing a MIMD Shared-Memory Computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[22] Mark A. Holliday. On the Effectiveness of Dynamic Page Placement. Technical Report CS-1989-19, Department of Computer Science, Duke University, Durham, NC, 1989.

[23] Mark A. Holliday. Reference History, Page Size, and Migration Daemons in Local/Remote Architectures. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, Boston, MA, April 1989. ACM/IEEE Computer Society.

[24] Herb Jacobs. A User-tunable Multiple Processor Scheduler. In *1986 USENIX Winter Conference Proceedings*, pages 183–191, January 1986.

[25] Anita K. Jones, Robert J. Chansler, Jr., Ivor Durham, Karsten Schwans, and Seven Vegdahl. StarOS, A Multiprocessor Operating System for the Support of Task Forces. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 117–127, December 1979.

[26] Paul A. Karger. Using Registers to Optimize Cross-Domain Call Performance. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–204, April 1989.

[27] Anna Karlin, Mark Manasse, Lyle McGeoch, and Susan Owicki. Competitive Randomized Algorithms for Non-Uniform Problems. In *Proceedings, Symposium on Discrete Algorithms, 1990*, San Francisco, CA, January 1990. ACM-SIAM.

[28] Anna Karlin, Mark Manasse, Larry Rudolph, and Daniel Sleator. Competitive Snoopy Caching. *Algorithmica*, 3(1):79–119, 1988.

[29] Lawrence J. Kenah and Simon F. Bate. *VAX/VMS Internals and Data Structures*. Digital Press, Maynard, MA, 1984.

[30] Leslie Lamport. Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[31] Leslie Lamport. Concurrent Reading and Writing of Clocks. Technical Report 27, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, 1988.

[32] Alan Langerman, Joseph Boykin, Susan LoVerso, and Shashi Mangalat. A Highly-Parallelized Mach-based Vnode Filesystem. In *Proceedings of the Winter 1990 USENIX Conference*, pages 297–312, January 1990.

[33] Richard P. LaRowe, Jr. and Carla S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. Technical Report CS-1990-10, Department of Computer Science, Duke University, Durham, NC, 1990.

[34] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley, Reading, MA, 1989.

[35] Charles Leiserson. Fat Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[36] Daniel Lenoski, Kouroush Gharachorloo, James Laudon, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. In *Digest of Papers, Spring Compcon '90*, pages 62–67, San Francisco, CA, February 1990. IEEE Computer Society.

[37] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *5th Symp. on Principles of Distributed Computing*, pages 229–239, 1986.

[38] Mark Manasse, Lyle McGeoch, and Daniel Sleator. Competitive Algorithms for On-line Problems. In *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pages 322–333. ACM SIGACT, May 1988.

[39] Elliot I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, 1972.

[40] G. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype: Introduction and Architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 764–771. IEEE Computer Society, 1985.

[41] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, Jonathan Chew, and William Bolosky. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Archtectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[42] Richard F. Rashid. Threads of a New System. *Unix Review*, 4(8):37–49, August 1986.

[43] Manfred Ruschitzka and R. S. Fabry. A Unifying Approach to Scheduling. *Communications of the ACM*, 20(7):469–477, July 1977.

[44] Christoph Scheurich and Michel Dubois. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. *IEEE Transactions on Computers*, 38(8):1154–1163, August 1989.

[45] Mark S. Squillante and Edward D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. Technical Report 89-06-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1990.

[46] Jeffrey H. Straathof, Ahok K. Thareja, and Ashok K. Agrawala. UNIX Scheduling for Large Systems. In *1986 USENIX Winter Conference Proceedings*, pages 111–139, January 1986.

[47] Avadis Tevanian, Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1987.

[48] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterwaithe, Jr. Firefly, A Multiprocessor Workstation. *IEEE Transactions on Computers*, 8(37):909–920, August 1988.

[49] Robert Thomas and Will Crowther. The Uniform System: An approach to runtime support for large scale shared memory multiprocessors. In *Proc. of 1988 Int. Conf. on Parallel Processing, Vol II*, pages 245–254, 1988.

[50] Hide Tokuda, James W. Wendorf, and Huay-Yeng Wang. Implementation of a Time-Driven Scheduler for Real-Time Operating Systems. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, December 1987.

[51] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[52] Wolf-Dietrich Weber. Personal communication. Conclusions from additional trace analysis of results for shared data.

[53] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[54] Mark Weiser, Alan Demers, and Carl Hauser. The Portable Common Runtime Approach to Interoperability. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 114–122, December 1989.

[55] James W. Wendorf. *Operating System/Application Concurrency in Tightly-Coupled Multiple-Processor Systems.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1987. Available as Technical Report CMU-CS-88-117.

[56] Andrew Wilson Jr. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Conference Proceedings, 14th International Symposium on Computer Architecture*, pages 244–252, Pittsburgh, PA, June 1987. ACM SIGARCH/IEEE Computer Society.

[57] William A. Wulf, Roy Levin, and Samuel P. Harbison. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill, New York, NY, 1981.

[58] Michael Young, Avadis Tevanian, Jr., Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *11th Symp. on Operating Systems Principles*, pages 63–76, 1987.

[59] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. In *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems*, December 1988. Proceedings published by North-Holland (1989), T. Hasegawa et. al. (eds.).

[60] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Processors. Technical Report 89-07-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1989.

[61] John Zahorjan and Cathy McCann. Processor Scheduling in Shared Memory Multiprocessors. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990.