EFFICIENT SCHEDULING ON MULTIPROGRAMMED SHARED-MEMORY MULTIPROCESSORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Andrew Tucker
December 1993

Abstract

Shared-memory multiprocessors are often used as compute servers, with multiple users running applications in a multiprogrammed style. On such systems, naive time-sharing scheduling policies, such as straightforward extensions of policies found on modern uniprocessors, can result in poor performance for parallel applications. The reason is that many parallel applications are written assuming a model where applications are running uninterrupted on a fixed number of processors. In compute server environments, where new applications are continuously moving in and out of the system, the computing environment is much more unstable; processes are frequently preempted to allow other applications' processes to run, and both the number of processors and the set of processors running an application varies over time. The result is a decrease in performance due to a variety of factors, including processes being obliviously preempted inside critical sections and cached data being replaced by intervening processes.

This thesis explores the problem of developing more sophisticated scheduling systems to avoid the performance problems raised by running parallel applications in a multiprogrammed environment. It begins with a discussion of the causes and extent of potential performance loss. It continues with a study of two previously proposed solutions, cache affinity scheduling and gang scheduling. Effective new implementations of each have been developed that maintain good response time and fair processor allocation. Performance results from a suite of multiprogrammed workloads containing parallel and serial applications, run on a 4-processor Silicon Graphics workstation, show improvements of up to 16% and 15% (over a standard UNIX scheduler) for cache affinity and gang scheduling, respectively.

The thesis then presents the design, implementation, and performance of a novel approach that offers high performance by combining the techniques of *process control* and *processor partitioning*. The process control approach is based on the principle that to maximize performance, a parallel application must dynamically match the number of runnable processes associated with it to the effective number of processors available. This avoids problems arising from frequent process preemption and allows applications to work at better operating points on their speedup curves. Processor partitioning is necessary for dealing with realistic multiprogramming environments, where both process controlled and non-controlled applications may be present. It also helps improve the cache performance of applications and removes the bottleneck associated with a single centralized scheduler. Performance results show speedups of up to 22% over the performance of a standard UNIX scheduler, even better than the 15–16% achieved by cache affinity and gang scheduling running the same workloads on the same machine.

Acknowledgements

Like any substantial body of work, this dissertation could not have been completed without help from many sources. I would first like to thank my advisor, Anoop Gupta, for his constant guidance over the past several years as I studied multiprocessor scheduling and performed the research and writing that resulted in this thesis. I would like to thank David Cheriton, Carl Gotsch, John Hennessy, and Mendel Rosenblum for serving as members of my oral defense committee. I would also like to thank David Cheriton and John Hennessy for serving as readers of my thesis, and making helpful comments that improved its quality.

My research was aided by many colleagues, both at Stanford and elsewhere. David Black, then at Carnegie-Mellon University, helped immeasurably with my early work with Mach, and contributed many valuable ideas. Locally, I have been helped by many of the members and associates of the DASH group. Luis Stevens and Josep Torellas worked closely with me in parts of this research. Luis also guided me through my first encounter with the SGI machine and its operating system. Rohit Chandra, Jonathan Rose, and Ed Rothberg provided substantial help with the applications used to measure performance. Jonathan Chew and Dave Nakahira helped with problems with the DASH machine.

Many others have made less tangible contributions to this thesis. I would like to thank my parents, Gilliam and Sandra Tucker, for their confidence of my ability to succeed in graduate school and their support when I needed it. I want to thank all of the friends who made graduate school a fun experience as well as an educational one, and the cycling buddies who gave me physical challenges to go along with the mental ones. Finally, thanks to the gang at Team Louis for helping me relax at the end of the day, and to Shandon Lloyd, for love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Although multiprocessors have become increasingly available in recent years, they still involve a substantial investment. This fact, combined with their considerable compute power, makes it attractive to use them as *compute servers*. Users running applications needing substantial CPU power would run them on the multiprocessors in preference to personal workstations. For ease of access and use, the parallel machines should be available to multiple users simultaneously, and be usable in a normal interactive manner. Thus, to work effectively in the role of compute servers the machines need to be able to efficiently handle a dynamic multiprogrammed load of applications that are both parallel and serial, CPU-bound and I/O bound.

The most common and easily-used type of multiprocessors are the shared-memory *bus-based* machines, which contain multiple processors communicating through a shared memory bus. These machines are also called UMA (Uniform Memory Access) machines, since access to a memory location via the bus takes the same amount of time regardless of which processor is performing the access and what memory location is being accessed. Cache coherency is maintained across processors through a variety of snooping and invalidation techniques. These machines are generally of small to moderate scale (usually less than 16–32 processors) due to problems with bus saturation with higher numbers of processors. For many uses, they provide a good compromise between high computational power, ease of use, and affordability. Examples include the Sun SPARCcenter 2000, the Silicon Graphics Power and Challenge machines, the Encore Multimax, and many others. The focus of this thesis is on finding the best operating system scheduling policies for these machines for multiprogrammed workloads.

1.1 Problem Statement

Applications written for parallel computers often assume that they will have sole use of the machine with all processors dedicated to them. This allows them to run at maximal performance when they do have the machine to themselves. However, on a multiprogrammed machine, where multiple users and applications may be active simultaneously, this is frequently not the case and each processor may be shared among multiple processes. In such environments, the throughput of the system can degrade substantially when the total number of active processes in the system

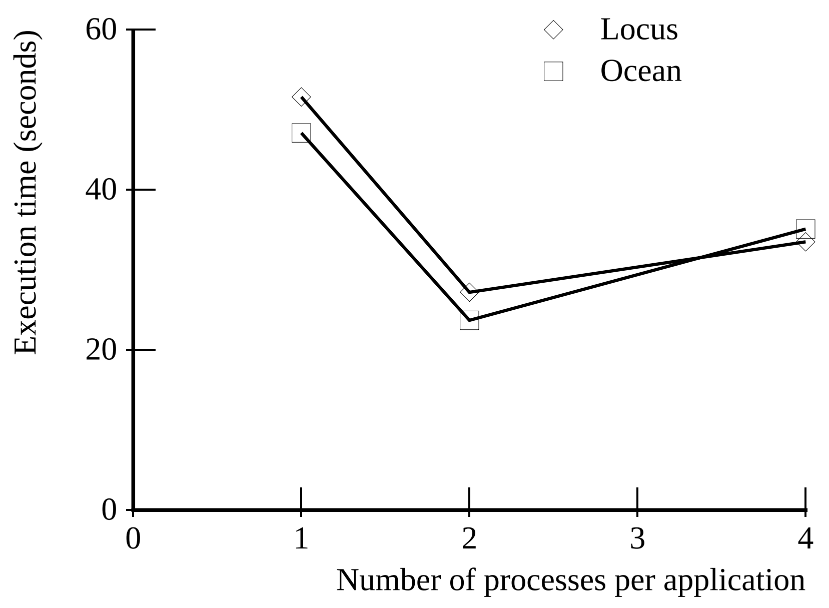

Figure 1.1: Execution time for `LocusRoute` and `Ocean` applications running simultaneously as the number of processes is varied. The applications were run on an SGI 4D/340 with 4 processors.

does not match the total number of processors.

Figure 1.1 shows the impact on performance when a parallel application's processes must contend for processors. The data is gathered from a Silicon Graphics 4D/340 multiprocessor with 4 processors. The graph shows the finish time for two parallel applications, `LocusRoute` and `Ocean`, when the two are started at the same time and as the number of processes is varied. `LocusRoute` [32] is a VLSI standard-cell router, and `Ocean` [37] is an ocean basin current simulator. (These applications, and the machine on which they are run, are discussed in detail in Chapter 2.) Each application breaks its problem into a number of tasks, which are scheduled onto the processes executing that application. The figure shows that the performance of both applications worsens considerably when the number of processes in each application exceeds two, and thus the total number of processes in the system exceeds four, the number of processors. Furthermore, the larger the number of processes, the worse the performance.

The performance degradation from multiprogramming, as seen above, can occur due to several reasons. First, there is the overhead of context switching between processes. Aside from the problem of corrupted caches (discussed below), a context-switch involves a number of system-specific operations, such as saving and restoring registers and switching address spaces, that do no real work. Second, many parallel applications use synchronization that requires busy-waiting on a variable. If the process that will set the variable is preempted, other processes may end-up wasting processor time waiting for that variable to be set [45]. Third, frequent context switching can indirectly affect processor cache behavior. When a context switch is performed, the preempted process may be rescheduled onto another processor, without the cache data that had been loaded into the cache of the original processor. Even if the process is rescheduled onto the same processor, intervening processes may have replaced needed cache data.

An additional factor can influence the performance of parallel applications in multiprogrammed workloads. Most parallel applications get sublinear speedups with increasing number of processors (i.e., the speedup with N processors is less than twice the speedup with $N/2$ processors) due to load balancing problems, worse cache behavior, greater contention for locks, etc. Consequently, when a large number of processors are used to run an application, the processor efficiency is less than when fewer processors are used. When a single application is running on a system, it generally gets best performance when its number of processes is equal to the number of processors in the system. When multiple applications are running simultaneously, however, only a few processes of each application are able to run at a time, and the applications will perform more efficiently if each uses fewer processes. We call this the *operating point* effect as it indicates a better "operating point" on the applications' speedup curves.

1.2 Previous Approaches

To address the above issues, several solutions have been proposed. One approach that has been used to address the problem of processes being preempted while inside critical sections is to use blocking or two-phase synchronization primitives instead of busy-waiting primitives [31, 18]. With blocking synchronization, processes yield the processor if they are unable to acquire a lock, allowing other processes to run and reducing the time until the process inside the critical section is rescheduled (assuming it is currently unscheduled). Two-phase synchronization involves spinning on the lock for a short duration before blocking, if the lock cannot be acquired. The idea is that if the process holding the lock is scheduled and running, it will soon release the lock, and the waiting process will be able to acquire it before yielding the processor. If the process holding the lock is unscheduled, the waiting process will block before wasting much processor time. Previous studies of this approach [18, 21] found it fairly effective and suggest that if the duration of spinning before blocking is fixed, the duration should be close to the duration of a context switch. More complex solutions have also been proposed involving adapting the spin duration to the specific lock in question based on previous history [21]. Other researchers have proposed keeping spinning synchronization but using intelligent schedulers that are given enough knowledge about the applications to avoid preempting processes while they are inside a critical section [13, 47].

1.2.1 Cache Affinity

The above approaches, while helping the problem of processes wasting time waiting for locks held by preempted processes, do not address the problem of cache behavior in a multiprogrammed system. In fact, they may worsen the problem, as blocking or two-phase synchronization can lead to increased context switching. As processors grow faster relative to memory speed, the effective use of data caches becomes more and more vital. To address the cache hit-rate problem, scheduling strategies that use *cache affinity* (the amount of relevant data a process has in some processor's cache) to determine which process to schedule on a processor have been proposed [41, 18]. Unfortunately, these techniques have had only limited success in real implementations [46].

One problem with cache affinity scheduling is that it is difficult to integrate into a general-purpose system where response time and fair allocation of processors are important. This thesis looks again at cache affinity scheduling, using a new implementation that avoids problems with response time and fairness. The performance of a number of multiprogrammed workloads improved by up to 10%, a small but significant increase.

1.2.2 Gang Scheduling

Researchers have also proposed the use of *gang scheduling* strategies [31, 13, 14, 8] that ensure that all processes belonging to an application execute at the same time. The idea is that by scheduling all processes simultaneously, synchronization performance will be improved. That is, time spent waiting for preempted processes (if using spinning synchronization) or excess context switching due to synchronization (if using blocking synchronization) will be reduced or eliminated.

Unfortunately, little work has been done on evaluating the performance of gang scheduling in a realistic environment. This thesis looks at the performance of gang scheduling when implemented on a real system, with varying results. The performance of some multiprogrammed workloads improved by as much as 15%, but other workloads slowed by up to 5%. On average, workload performance improved by 4%.

1.3 The Process Control Approach

The above scheduling approaches take as a given the number of processes in the system. They use scheduling tricks to avoid some of the problems associated with time-slicing between applications, but time-slicing is still prevalent and the problems are just reduced, not eliminated. They also do not consider or provide a solution to the operating point effect. In contrast to the above approaches, which focus almost exclusively on the operating system scheduler, we believe that a synergistic approach that involves both the application and the operating system can offer higher performance that resolves all of the performance problems resulting from multiprogramming. In this thesis, I propose and fully investigate one such approach called the *process control technique* [45]. It requires *process control* from the applications and *processor partitioning* and interface support from the operating system.

The process control technique is based on the principle that to maximize performance, a parallel application must dynamically control its number of runnable processes to match the effective number of processors available to it. In a multiprogramming environment, this adjustment of processes must be dynamic because other applications are constantly entering and leaving the system, and consequently the number of processors available to an application is constantly changing. By dynamically maintaining a match, context switches are largely eliminated and good cache and synchronization behavior can be ensured. The process control approach is most easily applied to the wide variety of parallel applications that are written using the task-queue or threads style [1, 6, 7, 11, 16, 19, 22], where user-level tasks (threads) are scheduled onto a number of kernel-scheduled server processes. In such an environment, the number of server processes can be safely changed without affecting the running of user-level tasks. Another important implication is that process control can be made totally transparent to the applications programmer by embedding it completely in the runtime system of a programming language or threads package.

In the processor partitioning technique, a policy module in the operating system continuously monitors the system load and dynamically (based on need, fairness, and priority) divides up the processors among the applications needing service. Scheduling of processes within a partition is handled at a lower level independently of the policy module. Processor partitioning is motivated by our need to handle realistic multiprogramming environments where we expect a mixture of applications—there may be some parallel applications that control their processes, there may be others that don't, and there may be single-process applications like compilers, editors, and network daemons. The problem with using process control in such environments, without processor partitioning, is that applications that do not control their processes may get an unfair share of the processors. The processor partitioning technique further allows a closer binding to be established between an application and the processors executing it, thus helping to improve cache performance. The technique also helps in removing the bottleneck associated with a single centralized scheduler in highly parallel machines.

In this thesis, these design, implementation, and policy issues involved in creating and using a process control system are studied in detail. Executions of parallel workloads using process control, on a 4-processor Silicon Graphics workstation, show an 8–22% performance improvement over the performance on a standard UNIX scheduler.

An approach similar to process control was developed concurrently with it at the University of Washington. This approach, *scheduler activations* [1], will be discussed in detail in Section 6.5.1, along with its differences from process control.

1.4 Thesis Contributions

This thesis addresses a number of issues concerning the problem of multiprogramming on multiprocessors. First, it studies two previously proposed approaches for resolving multiprogramming issues, cache affinity scheduling and gang scheduling, using new implementations that retain reasonable system response time and fairness of processor allocation in addition to effectively reaching the goals of the base scheduling policies. Both implementations involved additions to the standard UNIX priority system with a minimum of disruption to other parts of the operating system. The traditional application-kernel interface is also preserved.

Second, it develops process control, a novel synergistic approach to process scheduling involving cooperation between the application and kernel. This approach raises a large number of design and implementation issues, which I have considered and resolved so that process control can be reasonably used on real compute server systems.

Finally, it compares the performance of each scheduling approach, along with that of the "standard" UNIX scheduler, in a consistent and realistic manner. Each approach was tested running a variety of multiprogrammed parallel workloads, on the same high-performance multiprocessor running the same base operating system. It also studies the sources of the performance gains attained by each approach.

1.5 Thesis Organization

The thesis is structured as follows. Chapter 2 begins by considering the basic environment needed to provide a consistent basis for evaluating the performance of scheduling approaches. It discusses the machine to be used, the operating system that is modified to implement each new scheduling approach, and the applications and workloads that run on the system. It also specifies the experimental methodology used for gathering the data that is presented in the remainder of the thesis. Chapter 3 studies the performance loss that these workloads incur when running on the chosen machine using a standard UNIX scheduler, breaking down the performance loss according to source. This indicates the potential performance that each new scheduling approach might achieve.

Chapters 4 and 5 look at the performance of two previously proposed approaches to resolving problems with multiprogrammed parallel workloads, cache affinity scheduling and gang scheduling. In each case, the design and implementation issues of the approach are presented, followed by a description of our implementation, and finally a study of the performance results using the suite of workloads described in Chapter 2.

Chapters 6 and 7, present the process control approach. Chapter 6 presents the basic idea of process control and discusses the many design and implementation issues involved, and our solutions to those issues. Chapter 7 evaluates the performance of the process control approach. It also compares the performance of process control to that of the other proposed scheduling approaches.

Finally, Chapter 8 discusses how these results might extend to other types of machines, particularly with larger numbers of processors. It also discusses possible future work relating to this research, and more general directions of multiprocessor operating systems research.

Chapter 2

Experimental Environment

One of the main goals of this thesis, as discussed in Chapter 1, is to study and compare the performance of several different scheduling approaches in a multiprogrammed environment. Before we can do this, we need to determine how to measure the performance of different schedulers. The focus of this chapter is the environment used for performance evaluation experiments. I will first discuss the machine chosen and the base operating system that will be modified for each scheduling approach. I will then discuss the applications that will be executed and the workloads into which they will be grouped to provide a multiprogrammed load. Finally, I will briefly talk about the methodology used to gather results.

2.1 Base Machine and Operating System

To study scheduling performance for a class of machines, we would like to use a fairly typical representative of that class. The quantitative results presented in this thesis are based on a Silicon Graphics (SGI) PowerStation 4D/340 multiprocessor [3]. The system consists of four 33 MHz MIPS R3000/R3010 processors on a shared bus and provides peak computing power of about 100 MIPS and 35 MFLOPS. Each processor has a 64 Kbyte instruction cache, a 64 Kbyte first-level (primary) data cache and a 256 Kbyte second-level (secondary) data cache. A first-level cache hit costs 1 clock cycle (or 30 ns), a second-level hit costs about 14 clock cycles, and a second-level miss costs about 30 clock cycles. All caches are direct-mapped and have 16-byte blocks. (The machine has been slightly modified for use as a 4-processor cluster of the DASH distributed-memory multiprocessor [24]. An unmodified SGI 4D/340 has a second-level cache with a fetch size of 64 bytes and a miss cost of 50 cycles.) As a result of the large miss penalties, it is very important to have high cache-hit rates to get high processor utilizations. To accurately study the execution of applications, a performance monitoring board has been added to the system, providing a real-time clock with 60 ns granularity and hardware probes that trace cache misses.

The SGI machine is very typical of a modern bus-based shared-memory multiprocessor. The MIPS processors provide high performance at a reasonable cost. These machines are packaged as workstations and commonly sold for use as high-powered graphics engines, though their suitability for scientific computation is also readily apparent.

The operating system running on the SGI 4D/340 is IRIX, a multithreaded version of UNIX System V with added functionality for supporting parallel applications. The scheduler is a straightforward extension of the standard UNIX priority system to multiprocessor use. That is, runnable (but not running) processes are placed in a global queue, and each processor selects from the top of the queue. Processes are ranked by priority, and priority is derived from a base value plus the recent CPU usage of the process. Similar approaches are used in most UNIX systems [2, 23]. The IRIX system also includes some additional functionality, including an implementation of gang scheduling that will be discussed in Chapter 5.

2.2 Applications

Once we have a machine and operating system on which to implement our scheduling approaches, we need workloads to execute. Unfortunately, so far there has been very little work on standardized workloads for multiprogrammed use, and none using parallel applications. Most work on multiprogrammed parallel operating systems has either not done performance evaluation on a real machine or has used "toy" applications. Since either option is unsatisfactory, we have developed a set of workloads by combining complex and realistic applications of various types. These include both closely communicating parallel applications and essentially serial applications.

For the scheduling analysis, we primarily use workloads composed of five parallel applications selected from the SPLASH (Stanford ParalleL Applications for Shared Memory) suite of parallel benchmarks [38]. The SPLASH applications used in this thesis are `LocusRoute`, `Ocean`, `MP3D`, `Cholesky`, and `Water`. They are complex scientific and engineering programs representing the fields of VLSI design, oceanography, aeronautical simulation, numerical analysis, and molecular dynamics (described briefly below; more details can be found in the SPLASH report). The applications were chosen as a group of realistic scientific parallel codes, both "k-ernels" that make up integral parts of larger applications (`Cholesky`) and fully self-contained applications (`LocusRoute`, `Ocean`, `MP3D`, and `Water`).

In the standard SPLASH release of these applications (available by anonymous `ftp` from `mojave.stanford.edu`), parallelism is implemented using a collection of macros designed at the Argonne National Laboratory (ANL) [5]. However, in this thesis, alternate versions of the applications written in the COOL (Concurrent Object-Oriented Language) [6] parallel language are used. COOL provides basic tasking and synchronization mechanisms and an overall object-oriented framework for parallel programming. The basic mechanism is a collection of task queues containing the work to be performed in parallel. This will become important in Chapters 6 and 7, as a full implementation of process control requires applications that use task queues. The COOL versions of the SPLASH applications have been carefully optimized for running alone on the SGI multiprocessor so that their performance is comparable to the original (highly tuned) ANL macro versions. COOL uses distributed task queues to provide cache locality for applications, assuming its processes stay on the same processors. The synchronization primitives used to implement the COOL runtime system spin for a short period of time before blocking. COOL was chosen over other task-queue-based systems primarily because of familiarity with the internals of its runtime system and because several large parallel applications have been implemented using it;

other task-based parallel languages (such as Jade [22]) or even general-purpose user-level thread packages (such as C-threads [7]) would work as well.

One goal of this research was to test the effect of workloads including “serial” and I/O-bound applications on scheduling performance. A parallel version of the standard `make` application, `Pmake`, was selected as a commonly used program development tool that requires both good CPU and I/O performance to run well. `Pmake` is a component of the IRIX operating system used on Silicon Graphics machines.

We will now look at the purpose, structure, and memory requirements of these applications in a little more detail.

2.2.1 LocusRoute

`LocusRoute` [32] is a commercial-quality global router for VLSI standard cells. It uses an iterative algorithm to route wires with the goal of minimizing overall chip area. The application does this by selecting one wire to be routed at a time, and calculating a cost function for each possible route for that wire by counting the number of wires already routed in regions (or *routing cells*) that this wire would pass through. The wire is then given the route that incurs the lowest cost.

The major data structures consist of a shared cost array and data on wires’ pin positions and current routes. The cost array is frequently accessed and contains 8 bytes for each routing cell. The pin positions of the wires is roughly 300 bytes for each wire. Routes are stored in terms of 20-byte structures for each straight segment of the wire. The computation is broken into a number of independent tasks, each representing an unrouted wire. Each task calculates the cost function for all possible alternate routes for the wire and places the wire in the best route. The experiments in this thesis run `LocusRoute` with `Primary2`, a standard circuit containing 3817 wires, and perform 4 routing iterations. The cost array for `Primary2` contains 25800 routing cells (1290 in each of 20 channels), taking a total of 200 Kbytes of memory. The pin positions consume roughly 1 Mbyte. The routes are progressively stored as the application computes, finally amounting to about 600 Kbytes.

2.2.2 Ocean

`Ocean` [37] simulates eddy currents in an ocean basin using a discretized quasi-geostrophic circulation model. The simulation is performed for many time steps until the eddies and ocean flow attain a mutual balance. Each time step involves setting up and solving a set of spatial partial differential equations over two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin. The parallel version of the application solves the equations using the Gauss-Seidel method with successive over-relaxation (SOR).

The main data structures are 25 two-dimensional double-precision floating-point arrays holding the discretized values of the functions associated with the equations. The grid representing the basin is divided into tasks for each phase of the computation, and phases are separated by barrier synchronization. For the experiments, I used a 130×130 grid, with a tolerance for convergence of 10^{-7}. These parameters force a number of time steps resulting in a highly accurate simulation for problems of mesoscale (one to a few hundred kilometers) resolution. The active data set size with this grid size is about 3.2 Mbytes.

2.2.3 MP3D

`MP3D` [29] is a particle-based simulator used to study the pressure and temperature profiles created as an object flies at high speed through the upper atmosphere. It simulates the trajectories of a collection of representative of air molecules, subject to collisions with boundaries of the physical domain, objects under study, and other molecules. Space is represented by a three-dimensional *space array* of cells. Molecular collisions are statistically determined among molecules occupying the same cell. The overall computation consists of evaluating the positions and velocities of the molecules over a sequence of time steps and gathering appropriate statistics.

The primary data structures in `MP3D` are the state information of each molecule (36 bytes per molecule), and the space array cells (40 bytes per cell). Each time step is broken into five phases of execution, separated by barriers. Within each phase, parallel tasks are created by spatially partitioning the set of molecules. Hence, there is spatial locality in access to the molecules' data structures, but not to the space array. In these experiments `MP3D` is run with approximately 100,000 molecules for 10 time steps. A space array of $16 \times 16 \times 16 = 4096$ cells is used. Although a problem with 100,000 molecules is fairly small for an aerodynamic simulation, the amount of data accessed by each process still exceeds the capacity of both the first-level and second-level caches. With these parameters, the particles takes 3.4 Mbytes of memory, and the space array takes 160 Kbytes. Larger problems should behave similarly.

2.2.4 Cholesky

The `Cholesky` [33] application is used for factoring sparse, symmetric, positive definite matrices. Factorization of positive definite matrices is important in a number of domains including structural analysis and device and process simulation. Cholesky factorization solves the equation $A = LL^T$ for a lower triangular matrix L, given A by successively adding a multiple of one column to another column to cancel non-zeros in the upper triangle of the matrix.

In `Cholesky`, the dynamic supernodal fan-out method is used to parallelize the factorization method. The factorization is broken into tasks involving *supernodes*, groups of columns with similar non-zero structure. Large supernodes are further divided into *panels* to increase concurrency. A typical parallel task consists of one panel updating another panel in the matrix. The primary data structure is the matrix itself, stored by columns with arrays of non-zero elements and their corresponding rows. The experiments in this thesis used `bcsttk18`, a standard matrix from the Boeing/Harwell test set [12]. The matrix is fairly large, 11948×11948, with 68571 non-zero elements. The unfactored matrix takes about 1 megabyte; the factored matrix uses 5.7 megabytes. There are 7438 supernodes in the input matrix, of sizes up to 208 columns. The maximum panel size used in the computation contains 3240 non-zeros.

2.2.5 Water

`Water` [36] is an N-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. The computation performs a user-specified number of time steps, hopefully allowing the system to reach a steady state. Each time step involves setting up and solving the Newtonian equations for motion of water molecules in a cubical box with periodic boundary conditions.

The main data structure is an array of records, each holding all the data necessary for one molecule. Each record uses 600 bytes of memory. The work in each time-step is divided into phases, each a computation over all molecules in the system. Each phase is partitioned into tasks to be run in parallel by statically partitioning the molecules in the system. Barriers are used between the phases. The experiments use a problem with 343 molecules, running for 3 time steps. The small number of time steps allows us to evaluate a reasonable-size problem in a short amount of time; results should directly scale to runs with more time steps. With these parameters, the array of records takes 200 Kbytes of memory; if we include per-process private data structures, the entire application takes 230 Kbytes.

2.2.6 Pmake

`Pmake` is a parallel version of the standard UNIX `make` command [15]. It spawns multiple simultaneous compilation threads up to a predefined limit. The result is a number of simultaneously running I/O-dependent processes with very little intercommunication and synchronization. Due to the lack of communication, we can consider the separate processes as each being an essentially serial computation; by using multiple processes we simply reduce the amount of work done by each. The experiments use `Pmake` to compile a set of 17 independent C files (a portion of the IRIX kernel), averaging 800 lines of code each.

2.3 Application Performance

In order to get an idea of how the applications behave with different numbers of processes, and to show the effects of excess processes, we look at the performance of each application as the number of processes is varied from 1 to 16. The execution times of the parallel portion of the applications running on the 4-processor SGI machine are shown in Figure 2.1, and Figure 2.2 shows the corresponding speedup curves with each application normalized to the performance of the parallel code when run with one process. The parallel portion of `Pmake` is its entire execution.

With 1 to 4 processes, all five COOL applications behave similarly. The peak performance occurs when the number of processes matches the number of processors. The speedup with 4 processes for `LocusRoute`, `Ocean`, `MP3D`, `Cholesky`, and `Water` is 3.6, 3.3, 2.9, 3.0, and 3.9, respectively. The speedup is not perfect due to communication overhead, load balancing problems, redundant work done in multiple processes, and a worse cache hit rate as compared to the sequential runs. As the number of processes is increased, the cache hit rate gets worse due to two reasons: (i) there is less spatial locality as adjacent locations within the same cache line may be used by different processes (also called *false sharing*), and (ii) the number of misses corresponding to true communication between the parallel processes increases.

With greater than 4 processes, several of the COOL applications incur a significant drop

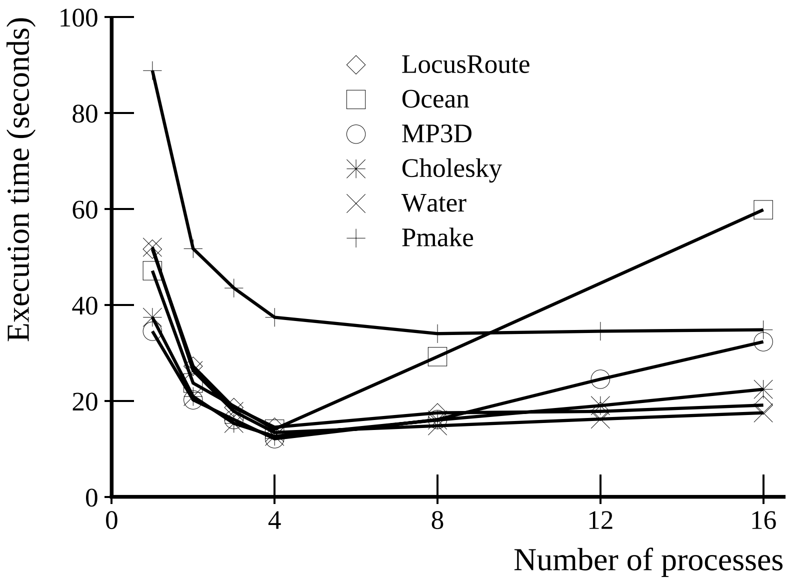

Figure 2.1: Execution times of parallel portions of applications as number of processes varies from 1 to 16, on a 4-processor SGI machine.

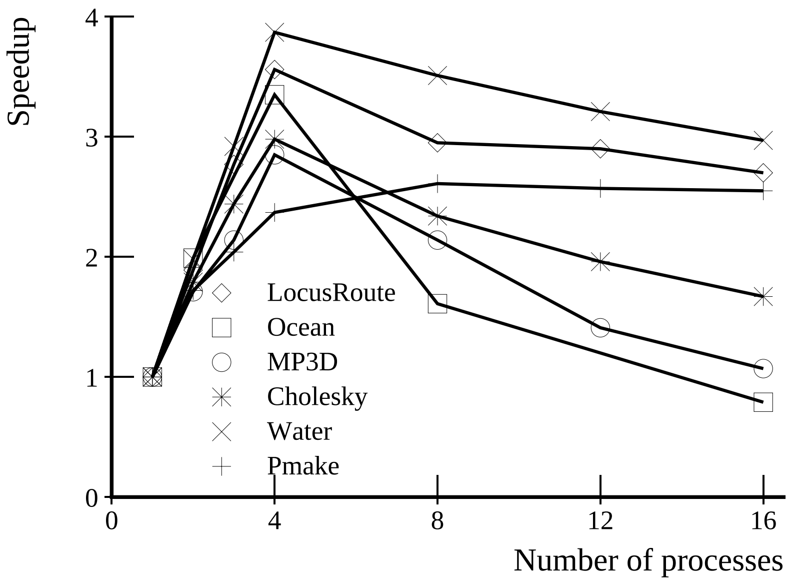

Figure 2.2: Normalized speedup curves of applications corresponding to the execution times in Figure 2.1.

in performance. Ideally, if there were no overheads to having more processes than processors available to run them, the curves would flatten out beyond 4 processes, achieving the same performance with 8, 12, and 16 processes as they did with 4 processes. Instead, `MP3D` and `Cholesky` take 1.3 times as long to run with 8 processes as they do with 4 processes. `Ocean` is even worse, taking over twice as long with 8 processes as it does with 4 processes. This is largely due to poor cache behavior as intervening processes flush relevant data from the cache. With 16 processes, the performance is even worse; `MP3D`, `Ocean`, and `Cholesky` slow down by factors of 2.7, 4.2, and 1.8, respectively, as compared to performance with 4 processes. `LocusRoute` and `Water` are somewhat less affected, both slowing by only a factor of 1.3 with 16 processes when compared with the 4-process performance.

`Pmake` is an exception to the rule that application performance drops when the number of processes exceeds the number of processors. Since compilations involve a large amount of I/O (reading in the file to be compiled and writing out the compiled object file), `Pmake` processes are often idle. Thus, running with only 4 processes leaves some processors idle, and the application is sped up by only a factor of 2.4. The result is that maximal performance is achieved with 8 processes, keeping more processors busy and reaching a speedup of 2.6 over the serial execution time. The experiments in this thesis run `Pmake` with 8 processes for this reason. The speedup is low even with 8 processes because of the substantial variation between the compilation times of different files, causing load imbalance, and because even with only one process some parallelism is achieved. This is because the compilation of a file invokes multiple child processes each representing a stage of the compilation—some of the stages can be slightly overlapped to achieve some parallelism even when multiple files are not being compiled simultaneously. The result is slightly better performance running with one process on a parallel machine that running with one process on an equivalent serial machine.

2.4 Benchmark Workloads

To evaluate the performance of different scheduling approaches, we have selected a number of workloads, made up of the above applications, to be used as a benchmark set. The workloads are listed in Table 2.1. The first column shows the names of the workloads, made up of the first initials of the constituent applications. Most of the workloads consist solely of the COOL applications, but two also include `Pmake`. In addition to these workloads, we will also use new workloads when necessary to examine or address a specific point.

2.5 Experimental Methodology

Workloads are executed with each application using 4 processes (matching the number of processors in the system), except `Pmake`, which uses 8 processes (as discussed earlier). This provides optimal performance for each application running alone. Although running with fewer processes might result in better performance when multiple applications are run simultaneously, applications cannot normally dynamically alter the number of processes they use to match the system load. For most experiments, the applications are synchronized before each enters its parallel section,

Table 2.1: Benchmark workload set for evaluation of scheduling policies.

	Applications					
Workload	LocusRoute	Ocean	MP3D	Cholesky	Water	Pmake
1 (LO)	√	√				
2 (LM)	√		√			
3 (OC)		√		√		
4 (OM)		√	√			
5 (LOC)	√	√		√		
6 (LMC)	√		√	√		
7 (LOP)	√	√				√
8 (LOMC)	√	√	√	√		
9 (LMCP)	√		√	√		√
10 (LOMCW)	√	√	√	√	√	

and the duration of execution of the parallel code is measured. This gives a good indication of the steady-state performance of various scheduling policies. The synchronization is necessary since the duration of serial setup code preceding the parallel section varies widely across the applications, and without synchronization some applications might finish executing before others begin executing in parallel.

Chapter 7 will also look at the performance of the workloads when no synchronization of parallel sections is performed, and the starts of the applications are staggered to provide a varying load. This will show the performance of the measured scheduling policies in a setting more closely resembling a real compute server load, where applications are constantly entering and leaving the system. These experiments will use the execution times of the entire applications (not just the parallel time) to give the “bottom-line” performance of the applications (i.e., the time a hypothetical user would have to wait for his or her application to finish).

Performance measurement of operating systems is difficult because they run in a very heterogeneous environment. Network traffic, system daemons, extra applications, and many other things can affect performance, possibly without the knowledge of the experimenter. For this reason, I chose to isolate the system I was using to a limited extent, and used multiple runs to account for statistical variations. All data in this thesis was gathered with only the system console active, and all other access to the machine locked out. While it remained attached to the local network, no other machines used its files remotely. Each data point was gathered by running the experiment 10 times and averaging the results after examining them for anomalies (such as might occur due to a transient high network load, for example).

Chapter 3

Sources of Performance Loss

Chapter 1 introduced the possible sources of performance loss occurring in multiprogrammed systems. In this chapter we discuss these sources in more detail and study the real effect on performance of each source. The sources discussed in Chapter 1 were direct overhead from context switching, increased synchronization cost, and cache data replacement cost. Sections 3.1–3.3 consider each source in turn, estimating the effect on the performance of the workloads and machine proposed in Chapter 2. In Section 3.4, we consider the influence the "operating point effect" has on performance. Section 3.5 estimates the potential performance that might be achieved if the multiprogramming problems were eliminated, and applications could be run at their optimal operating points. In later chapters, these estimates will be used to evaluate the effectiveness of real scheduling approaches that propose to eliminate one or more of these sources of performance loss.

3.1 Direct Context Switch Overhead

The simplest source of performance loss in a multiprogrammed system is the amount of time taken to store the state of a preempted process and retrieve the state of a newly scheduled process. However, this time is usually fairly low in relation to its frequency. Table 3.1 shows the average duration of *dispatch intervals* for the workloads proposed in Chapter 2, run on the SGI machine also discussed in Chapter 2. A dispatch interval is the duration a process runs continuously on a processor and corresponds to the interval between context switches. The SGI machine uses a typical default time quantum of 30 milliseconds, so no dispatch interval is longer than this value. The table also shows the direct cost of the context switches in terms of percentage of performance lost. The SGI machine takes about 50 microseconds to switch processes. The result is a direct overhead from context switching that varies between 0.7–2.9%. As one may have expected, the workloads with the most overhead are those including the `Pmake` application. The mean overhead is only 1.3%. From this data, we can see that the direct cost of context switching is a small part of any performance loss from multiprogramming.

Table 3.1: Mean dispatch interval duration for the workloads, and estimated cost of context switching.

Workload	Mean Dispatch Interval (ms)	Cost of Context Switches (%)
LO	4.8	1.0
LM	5.3	0.9
OC	3.8	1.3
OM	7.0	0.7
LOC	4.3	1.2
LMC	5.4	0.9
LOP	1.9	2.6
LMCP	1.7	2.9
LOMC	5.0	1.0
LOMCW	4.4	1.1

3.2 Synchronization Effects

The second potential source of performance loss is poor synchronization behavior. As discussed in Chapter 1, synchronization primitives used by applications (such as acquiring a mutual exclusion lock) can generally be characterized as either *spinning* or *blocking*. The former type busy-wait if unable to successfully perform the synchronization; the latter yield the processor, allowing other processes to perform useful work on the processor even if the synchronizing process must wait. A slight modification of the blocking strategy is *two-phase* synchronization, where processes attempting to synchronize first busy-wait for a period of time (the "spinning" phase), then block if still unable to perform the synchronization (the "blocking" phase).

Parallel application programmers often use spinning synchronization because it involves minimal overhead. For example, if a lock will become available in a few microseconds, it is much more efficient for a process trying to acquire the lock to busy-wait rather than incurring the cost of a context switch. Since parallel applications are tuned to minimize synchronization costs, the amount of time spent waiting for synchronization can generally be assumed to be low. However, this tuning generally assumes a uniprogrammed environment, where each processor runs only a single process.

When run in a multiprogrammed environment, performance in the presence of spinning synchronization can worsen considerably. In a multiprogrammed environment, processes are frequently preempted to allow processes from other applications to run. If a process is preempted at the wrong time, other processes of the same application may be forced to wait until the preempted process is rescheduled to complete synchronization. For example, consider a process preempted while inside a critical section controlled by a mutual exclusion lock. Other processes, trying to enter the critical section, are forced to wait. The result is similar for barrier synchronization, where other processes in the application are forced to wait for a preempted process to complete its part of the barrier-controlled computation. With spinning synchronization, the time spent waiting is wasted processor time and results in a reduction in performance. Simulation studies, done jointly by the author along with Anoop Gupta and Shigeru Urushibara, found that parallel applications using spinning synchronization in a multiprogrammed environment often spend over 50% of their execution time waiting for preempted processes [18]. Another earlier work by the author and Anoop Gupta found that reducing context switching by using process control improved the performance of multiprogrammed applications using spinning synchronization by over 200%, largely by reducing the amount of busy-waiting.

A partial solution to this problem is to use blocking or two-phase synchronization. Waiting processes may then yield their processors, either immediately after unsuccessfully trying to synchronize or after a short spinning phase. This allows other processes and applications to do useful work while the synchronizing processes wait for a preempted process to be rescheduled. Yielding the processor can also reduce the amount of time until a preempted process is rescheduled. Strict blocking synchronization can add unwanted overhead when busy-waiting for a short period of time is cheaper than a context switch. Two-phase synchronization, however, adds little cost unless the process is forced to wait for more than a few microseconds. In the simulation study mentioned above, two-phase synchronization reduced synchronization time to less than 4%.

The parallel applications in the suite of workloads introduced in Chapter 2 use two-phase locks for these reasons, and the effects of multiprogramming on synchronization performance in these applications is thus less than if spinning synchronization were used. Synchronization is also reduced due to the use of a task-queue-based language for the implementation of the COOL applications, since application synchronization is often handled at a task level by blocking a task rather than the process running that task. Synchronization required by the COOL runtime system, however, must block processes, and synchronization time is still increased in the presence of multiprogramming.

Figure 3.1 shows the amount of time (in milliseconds) each workload spends spinning waiting on synchronization, broken down by application. The left bars (labeled “U”) show the time when the workload is run one application at a time (uniprogrammed rather than multiprogrammed). The right bars (labeled “M”) show the time when all applications are run together in multiprogrammed fashion on the IRIX system. Since the applications use two-phase synchronization, the spinning time is fairly low, as processes spin for only a short time before yielding the processor. Figure 3.2 shows the corresponding data, with the same format, for the number of times the applications in each workload yield the processor due to synchronization.

From the figures, we see that the time spent spinning in each application has little effect on the performance of the workload, due to the use of two-phase synchronization, and is often even reduced when multiple applications are run simultaneously. At most about 100 milliseconds is spent spinning, corresponding to less than 0.4% of the execution time of any of the workloads. However, the number of times a process yields the processor because it cannot acquire a lock can have a larger effect on performance, and can increase substantially. For example, workload LOC has a 27% reduction in spinning time when the applications are multiprogrammed, but a 316% increase in the number of process yields. (Application `Cholesky` requires the bulk of the synchronization due to short tasks with a high degree of dependency; workloads that do not include `Cholesky` (LO, LM, OM, and LOP) are less affected by synchronization problems.)

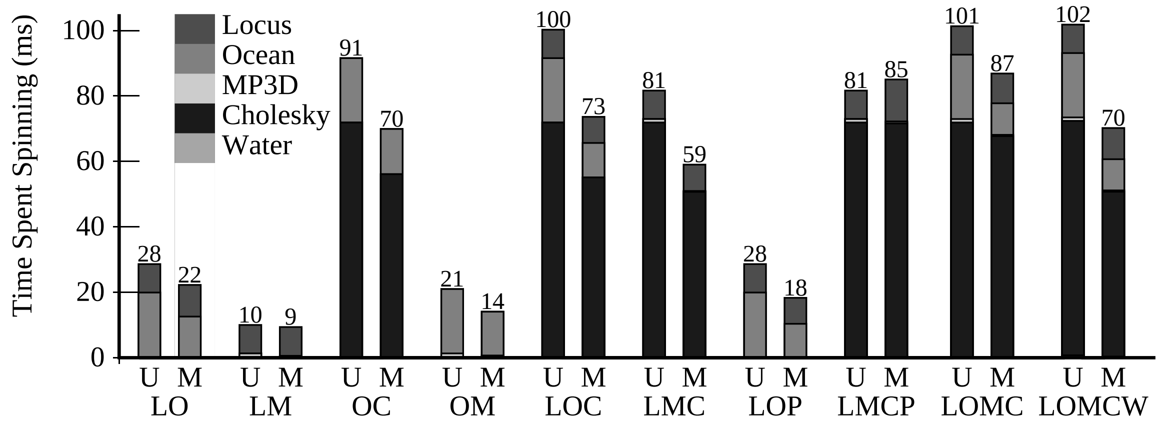

Figure 3.1: Total time (in milliseconds) spent spinning waiting for synchronization in each workload, both running one application at a time (U) and with all applications running together (M).

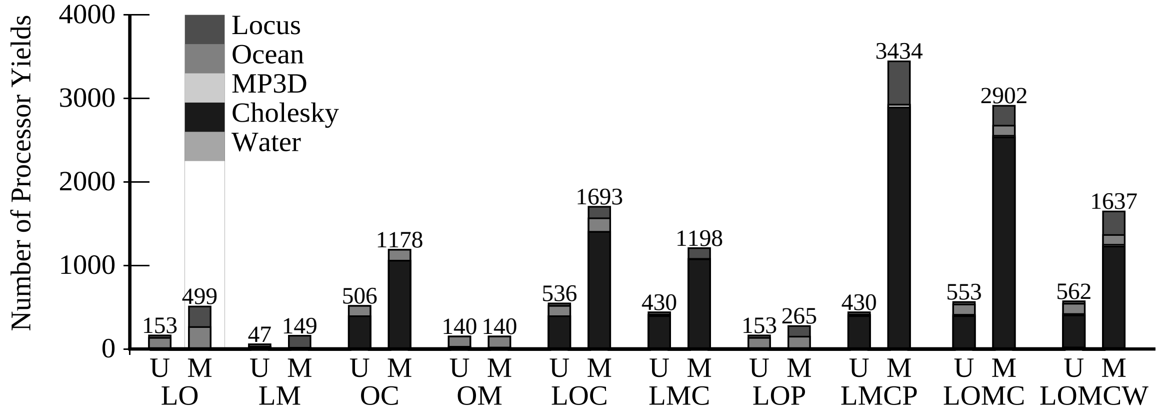

Figure 3.2: Number of times processes yield the processor due to synchronization in each workload, both running one application at a time (U) and with all applications running together (M).

Each time a process yields the processor the result is a context switch. The previous section showed that the direct cost of context switches, even under multiprogramming, is low (less than 2% of execution time). However, an increase in context switches can result in increased loss of cached data. This indirect effect, to be discussed further in the next section, can have a much higher impact on performance. Hence the main effect of synchronization problems incurred by multiprogramming results from the increase in context switches when processes reach the end of the spinning phase of their synchronization. Note that with spinning locks, there would be no increase in context switches, but the time spent spinning would be much longer.

3.3 Cache Data Loss

The third factor we have mentioned is the problem of previously cached data being unavailable when a process is scheduled. When a process is unscheduled, due to blocking on I/O, time quantum expiration, or yielding the processor, a context switch occurs and another process executes. The unscheduled process may next be scheduled on another processor, forcing data used by the process to be fetched into the new processor's cache (and, if modified, invalidated in the old processor's cache). Even if the process is next scheduled on the processor it had been using, intervening processes may have replaced its data from the cache. This can have a serious impact on cache performance.

We can estimate the performance lost due to cache data loss by measuring the amount of cached data reused by each application across successive time slices. This gives us an upper limit on the increase in cache misses, assuming none of a preempted process's data is in the cache when the process is rescheduled. By measuring the duration of the application's dispatch intervals, we can then estimate the effect of the additional misses on performance.

We used a special performance monitor board on our system to measure the misses in each application, both running normally and running while flushing the cache after every dispatch interval. The difference between the two shows the amount of data reused in the cache. The applications used two-phase locks to minimize spinning due to synchronization. The first column of Table 3.2 shows, for each application, the difference in the number of cache misses (or the number of reused 16-byte cache lines) per dispatch interval. The second column shows the mean duration of dispatch intervals for each application running alone. The third column shows the amount of time taken to fetch the reused data each dispatch interval divided by the duration of the dispatch interval, giving an upper limit on the performance loss incurred by cache data loss in a multiprogrammed environment. The result is an upper limit because it assumes that all of the data used by a process is unavailable when the process next executes; this is not always true. However, it is often close to the truth, particularly when the process is scheduled on another processor, or an intervening process "walks" through the cache replacing all of the data (common in applications that sequentially access large contiguously-allocated data structures).

We will now consider the data for each application in turn. `LocusRoute` has a small amount of reused data, 200 cache lines per dispatch interval (corresponding to 3200 bytes). However, the dispatch intervals are also short, an average of 1.7 ms. The result is a potential performance loss of about 11%, assuming each miss must be filled from memory (recall that the cycle time is

Table 3.2: Amount of reused data and mean dispatch interval for the applications.

Application	Reused Data per Dispatch Interval (misses)	Mean Dispatch Interval (ms)	Potential Performance Loss (%)
`LocusRoute`	200	1.7	10.7
`Ocean`	2200	9.5	21.1
`MP3D`	0	7.9	0.0
`Cholesky`	40	1.6	2.3
`Water`	2500	21.8	10.4
`Pmake`	500	2.0	22.7

30 ns and the latency of a miss is 30 cycles) rather than the second-level cache. `Ocean` has a much larger amount of reused data per dispatch interval (2200 cache lines or about 35 Kbytes), but also longer dispatch intervals (9.5 ms). The potential performance loss in `Ocean` is about 21%. `MP3D` has essentially no cache data reuse, as the data set for each process is generally much larger than the size of the cache. `Cholesky` also has a small amount of reused data per dispatch interval but has an even smaller amount of reused data (40 cache lines) with similar-length dispatch intervals, for a potential performance loss of 2%. Water is similar to `Ocean`, with 2500 cache lines reused across dispatch intervals, but a longer average dispatch interval (21.8 ms). The potential performance loss is about 10%. Finally, `Pmake` has a moderate amount of reused data (500 cache lines or about 8 Kbytes) and short dispatch intervals (2 ms). The result is a potential performance loss of 23% due to cache data loss under multiprogramming.

The `Pmake` case demonstrates that even serial applications can be affected by multiprogrammed workloads on a parallel machine. This problem can also also occur on a serial machine, though without the process migration component. Previous researchers have investigated the effect of context switching on the performance of applications on a serial machine, both using analytical techniques and using memory access traces of real executions [30, 43]. They found that context switching can have a significant effect on performance even in serial environments.

The above data on individual applications can be used to estimate the effect of cache data loss on multiprogrammed workloads. If we assume that none of the cache data used by a process in a dispatch interval remains in the cache the next time the process runs on the same processor, we can calculate the performance loss in each workload by averaging the losses from the constituent applications. Table 3.3 shows the results of these calculations. The result is an upper limit on the cost of cache data loss that varies between 4–18% for these workloads. The mean cost is 10%. Although the real cost will be lower (since previously cached data is not always replaced when a process is rescheduled), this suggests that the problem of cache data loss is much more serious than the direct cost of context switching.

As mentioned in the last section, synchronization problems can contribute to the problem of cache data loss by increasing the context switch rate. We now consider how much of the

Table 3.3: Performance cost for each multiprogrammed workload resulting from cache data loss, using a standard UNIX scheduler.

Workload	Cache Data Loss (%)
LO	16
LM	5
OC	11
OM	11
LOC	11
LMC	4
LOP	18
LMCP	9
LOMC	8
LOMCW	9

performance loss just discussed can be attributed to synchronization effects. Figure 3.2 showed, for each multiprogrammed workload, the increase in the number of times a process yields the processor (causing a context switch) because it cannot successfully synchronize. Table 3.4 shows the percentage change in the context switch rate this induces. The numbers range from 0–21%, and most are below 10%, suggesting that scheduling strategies that attempt to resolve problems with synchronization (such as gang scheduling, to be discussed in Chapter 5) will have only a small effect on cache data loss.

3.4 Operating Point Effect

The final performance issue that we consider is the operating point effect. As discussed in Chapter 1, the efficiency of a parallel application normally decreases as the number of processes (and processors) running the application is increased due to a number of problems including load balancing, decreasing spatial locality, communication overhead, and redundant work. The effect of this is that in a multiprogrammed workload, where each application receives only a portion of the total processor time, applications will run more quickly if they have fewer processes.

For example, consider workload LO, consisting of a `LocusRoute` application and an `Ocean` application. Normally, this workload would be run with each application using four processes. However, since the four processors of the machine are shared among two applications, each application generally runs on only two processors at a time. If each application were run with two processes, performance would be improved. Although the amount of processor time received by the applications would be the same, the applications would run more efficiently due to the reduced number of processes. They would be running at a better *operating point* on their speedup curves.

Table 3.4: Increase in context switch rate of multiprogrammed workloads resulting from use of two-phase synchronization, when using a standard UNIX scheduler.

Workload	Change in Context Switch Rate (%)
LO	5
LM	2
OC	9
OM	0
LOC	12
LMC	11
LOP	0
LMCP	9
LOMC	21
LOMCW	7

Table 3.5 shows the efficiency of each application when running with 2 and 4 processes. The efficiency with 2 and 4 processes was calculated by dividing the speedup obtained with 2 and 4 processes by 2 and 4, respectively.

Table 3.5: Efficiency of applications when running with two and four processes, compared with “ideal” performance based on execution times running with one process.

Application	Efficiency (%)	
	2 processes	4 processes
`LocusRoute`	95	89
`Ocean`	99	84
`MP3D`	86	71
`Cholesky`	90	75
`Water`	99	97
`Pmake`	86	59

The table shows that all of the applications but `Water` suffer substantial losses in efficiency with four processes. With two processes, the situation is better, though `MP3D`, `Cholesky`, and `Pmake` still lose 10% or more. From this data we can conclude that if applications were to run with two processes rather than four when only half of a 4-processor machine was available (as when two parallel applications are running simultaneously), their performance would improve by 2–27% (the difference between the efficiency with two processes and that with four processes). If they were to be run with one process rather than four when only a quarter of a 4-processor machine was available (as when four parallel applications are running simultaneously), their performance would improve by 3–41% (the difference between “ideal” or 100% efficiency and that with four processes).

In conclusion, the operating point effect has the potential to substantially impact the performance of parallel applications. Although not a direct “cost” of multiprogramming, the operating point effect indicates that applications running without concern for the number of processes they use can be running much slower than their potential in a multiprogrammed environment. In the next section, we will hypothesize a scheduling method that runs each application at its optimal operating point at any given time. This method would also eliminate performance losses from multiprogramming due to synchronization and cache data loss. The result will be optimal performance for multiprogrammed workloads. We will estimate its effect on our set of workloads to determine the extent to which appropriate scheduling techniques can improve multiprogrammed performance.

3.5 Upper Bound on Achieved Performance

The previous sections have examined potential sources of performance loss from multiprogramming and looked at the impact of each source on application and workload performance. This section will consider the question of the overall impact of all sources of performance loss; that is, how much performance is being lost due to all of these problems combined? The results will indicate the potential for any new scheduling policies to improve performance.

Figure 3.3 shows this effect for our suite of multiprogrammed workloads. Two sets of execution times are shown for each of our workloads, both normalized with respect to the performance of the workload running under a standard UNIX scheduler. The full length of the bars (“batch”) indicates the potential performance of the workloads when the costs of multiprogramming (not including the operating point effect) are removed. Each application in a workload is run in batch style with four processes, using the entire system until completion; the next application in the workload is then executed. Although this is not a reasonable approach for scheduling applications on a real system, due to considerations of fairness and response time, it is useful for analyzing the costs of multiprogramming. Since each application has sole use of the system when it is executing, no context switching occurs, and the direct and indirect costs associated with context switching, including time taken to perform the context switch, time waiting to synchronize with preempted processes, and cache data loss, are eliminated. The execution times of the parallel portions of the applications are then added to provide a comparison with their execution times running on a standard UNIX scheduler, where the system must be shared among applications to ensure fair allocation of processor time and reasonable response time.

A caveat for this data is that in comparing executions where the number of processes does not exceed the number of processors (e.g., the “batch” model and the “operating point” model discussed below) to executions where the number of processes exceeds the number of processors (e.g., the standard UNIX scheduler results to which the others are normalized), we can miss factors that may actually *improve* performance under multiprogramming. For example, when the number of processes exceeds the number of processors, and a process blocks on I/O, processes

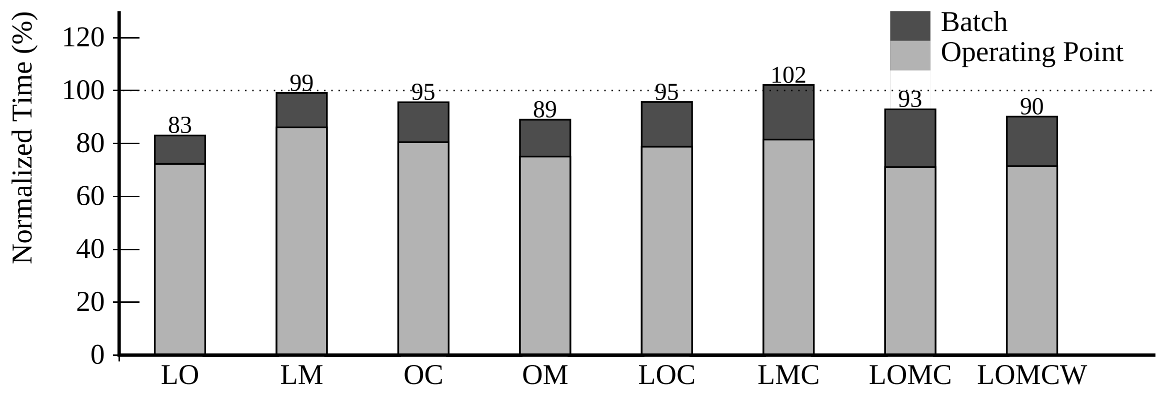

Figure 3.3: Potential performance of workloads, shown using batch mode with 4 processes per application ("batch"), and limiting the number of processes of each application to its optimal operating point ("operating point"). Execution times are normalized with respect to performance with standard UNIX scheduling.

of other applications may run in its place. Workload LMC actually appears to perform slightly (2%) worse in the "batch" model than with a standard UNIX scheduler. Although the limited amount of I/O used by the applications tends to minimize these problems, scheduling policies that ensure that processors do not sit idle while processes block on I/O may achieve a slightly higher performance gain than indicated in Figure 3.3.

The light-shaded parts of the bars in the figure ("operating point") correspond to data derived in a more complicated fashion. The idea is to not only eliminate the costs of multiprogramming, as above, but also take advantage of the operating point effect by executing each application with a number of processes equal to the number of processors available to run that application. This ensures that each application will use the available processors as efficiently as possible. Thus, a workload with two applications, such as LO, executes each application with two processes. A workload with four applications, such as LOMC, similarly executes each application with one processes. LOMCW, with five applications, also runs with one process per application, since a lower number is not possible. Workloads with three applications are more of a problem, however. In these workloads, such as LOC, we can run one application (say `LocusRoute`) with two processes and the other two with one process. However, this results in an unfair allocation of resources. In addition to the unattractiveness of an unfair policy, it results in a performance problem. The `LocusRoute` application finishes very quickly, using two processes, while the others take much longer, with one process each. After `LocusRoute` finishes, two of the processors on the machine are idle, while `Ocean` and `Cholesky` applications remain running. This can happen even in workloads with two or four applications, if one application finishes executing before others.

The solution to these problems is to consider the number of processes used by an application to be *dynamic*. For a workload like LO with two applications, each application is started with two processes. When one application finishes executing, the other then runs with four processes to take advantage of the extra processors that have become available. Similarly, a three-application workload like LOC starts running with `LocusRoute` using two processes and `Ocean` and `Cholesky` using one each. After a short interval, `LocusRoute` drops to using one process and `Ocean` uses two processes. After an equal-length interval, `LocusRoute` and `Ocean` run with one process each and `Cholesky` uses two processes. Finally, the two process usage cycles back to `LocusRoute` again, and so on until one of the applications finishes executing. The result is to execute the applications at the most efficient points possible while still maintaining fair allocation of processor time.

An interesting and important effect of executing workloads in this manner is the virtual elimination of context switching. Note that in each workload above, the number of processes used by each application is adjusted so that the total number of processes in the system matches the number of processors. Hence, there is no need to share a processor among multiple processes, and context switching is not needed. As a result, the dark-shaded part of the bars in the figure, indicating the difference between the "batch" data and the "operating point" data, shows the impact of the operating point effect. The process control scheduling approach mentioned in Chapter 1 uses the same basic idea to eliminate context switching and run applications at their optimal operating point. Chapter 6 will discuss in detail the design and policy issues involved in implementing this approach in a real general-purpose system environment.

Figure 3.3 shows the performance after removing multiprogramming costs, indicated by the full bars, to be slightly better than the performance of the standard UNIX scheduler. The geometric mean of the normalized workload execution times is 93.1%, a 6.9% improvement over the standard scheduler. This suggests that scheduling policies addressing these costs may be able to improve performance by a small but significant amount. Note that this indicates that the estimates of the performance effects of cache data loss in Section 3.3 were slightly excessive; the assumption that no previously cached data remains in the cache after a process is rescheduled was overly pessimistic.

The impact of the operating point effect is substantially greater. The geometric mean of the workloads in the light-shaded parts of the bars is 76.6%, a 23.4% improvement over the standard UNIX scheduler and a 16.5% improvement over the full bars. The operating point effect is thus shown to be more significant than the direct costs of multiprogramming identified in Sections 3.1–3.3. We shall see in future chapters that while scheduling policies that address only the direct costs of multiprogramming can improve performance slightly, a scheduling policy that also takes advantage of the operating point effect performs much better.

The careful reader will have noticed that Figure 3.3 only includes 8 workloads out of our suite of 10; workloads LOP and LMCP are missing. The reason for this is those workloads include the `Pmake` application. Unfortunately, since `Pmake` contains a large amount of I/O, it is not conducive to the sort of modeling performed for the data in the figure. In the case of the "batch" model, `Pmake` runs almost as fast when running with other applications as it does running alone. The result of the model would show the missing workloads running much slower in batch mode than they do with a standard UNIX scheduler. In the case of the "operating point" model, `Pmake` varies the number of processes it uses continually. While a more sophisticated system (such as the process control system described in Chapter 6) will adjust to this, it is difficult to manage with the simple calculations performed in this section.

3.6 Conclusions

This chapter has discussed the possible sources of performance problems when applications are run in a multiprogrammed style, and the extent to which each source contributes to overall performance. These sources can be divided into two kinds: direct costs, resulting from context switching between multiple applications contending for processors, and an indirect cost, resulting from running applications with a sub-optimal number of processes given the number of processors available.

The first kind of performance loss includes the direct cost of context switching, synchronization problems, and cache data loss. The direct cost was shown to be relatively insignificant, counting for an average of less than 2% of execution time. Additionally, the increase in time processes spent busy-waiting for synchronization due to multiprogramming in our applications was also low, as the applications use two-phase locks to minimize busy-waiting. However, using two-phase locks in the presence of multiprogramming also increases the number of times processes yield the processor, raising the context switch rate and worsening problems associated with context switching. The performance cost of cache data loss is even more significant, potentially slowing workloads by as much as 18%. We expect that scheduling policies that resolve either cache data loss or synchronization problems may provide a small but significant benefit to system performance under multiprogramming.

The second, indirect, kind of performance loss is associated with the "operating point" effect, where most applications run more efficiently with fewer processes. By dynamically adjusting the number of processes an application uses in order to run with maximal efficiency, we can obtain a substantial performance advantage. A scheduling policy that works in this manner can attain an average performance advantage of 17% from the operating point effect and an additional 6% by reducing context switching and minimizing synchronization problems. Policies only reducing context switching and minimizing synchronization problems will only be able to achieve the 6% advantage.

The next four chapters will describe policies to address these problems. Cache affinity scheduling, discussed in Chapter 4, tries to reduce cache data loss due to context switching. Gang scheduling, discussed in Chapter 5, tries to minimize synchronization problems. Finally, process control, discussed in Chapters 6 and 7, tries to do both of these in addition to taking advantage of the operating point effect.

Chapter 4

Cache Affinity

The performance achieved by applications on shared-memory multiprocessors is often highly sensitive to the latency of memory accesses. In small- and medium-scale high-performance machines, this latency can often be several tens of cycles. Consequently, it is important to develop techniques to reduce the number of cache misses suffered by the workloads running on these machines.

There are two factors that increase the number of misses when running multiprogrammed workloads. First, since an idle processor simply selects the highest priority runnable process, a given process often moves from one CPU to another. This frequent migration results in the process having to continuously reload its state into new caches, producing streams of cache misses. Second, several processes are forced to time-share the same cache, resulting in one process displacing the cache state previously built up by an earlier one. Consequently, even if this earlier process does not migrate, when it is next scheduled it will generate a stream of misses as it rebuilds its cache state.

To reduce the number of misses in these workloads, processes should reuse their cached state more. One way to encourage this is to schedule each process based on its affinity to individual caches, that is, based on the amount of state that the process has accumulated in an individual cache. This technique is called *cache affinity scheduling*.

Cache affinity scheduling has been the subject of several previous studies. However, many of these studies were based on analytical modeling [41], simulations [18], or synthetic applications [10], missing the subtle effects of real applications running on a real machine. Others studied uniprocessors [30], single applications within a narrow domain [42], or systems with unusual (space-sharing) scheduling policies [46].

This chapter makes a more thorough attempt to understand the effect of cache affinity scheduling on a real system running a variety of applications. It first investigates the ways application and workload characteristics affect their ability to benefit from affinity scheduling. Section 3.3 used the cache data reuse and mean dispatch interval length of each application to estimate the performance lost due to cache data replacement or loss. This chapter extends this work by also using the amount of process blocking and the effect of other applications in the workload to estimate the extent to which cache affinity scheduling can improve performance. We use hardware probes into the SGI 4D/340 to measure these characteristics for our suite of multiprogrammed workloads and find that workloads must satisfy a number of conditions to achieve even moderate gains from affinity.

We then compare our model with the performance of the workloads running with a simple but effective affinity scheduler on the SGI machine. Although the affinity scheduler does improve the performance of most of the workloads, the gains are at most only 10%. We compare the results with two alternative methods that have been used to exploit cache reuse, namely attaching processes to processors and extending the time quantum, and finally discuss the benefits of more complex ways of implementing affinity scheduling.

The contents of this chapter are based on previously published work done jointly by the author along with Josep Torellas and Anoop Gupta [44]. The older work developed a workable implementation of affinity scheduling and applied it to a number of workloads consisting of parallel applications using ANL macros [5] to provide explicit parallelism. This chapter re-applies affinity scheduling to the workloads selected for this thesis so that its performance can be compared with that of gang scheduling and process control.

This chapter is organized as follows. Section 4.1 characterizes the workloads from the s-tandpoint of cache affinity. Section 4.2 describes the baseline affinity function. In Section 4.3, we discuss the results, first focusing on the basic results of affinity scheduling (Section 4.3.1), and then on attached scheduling and increasing the time quantum (Section 4.3.2 and 4.3.3). We present discussion and related work in Section 4.4, and conclude in Section 4.5.

4.1 Potential Benefits of Cache Affinity

There are numerous factors that can affect the performance of a workload using cache affinity scheduling. One of the purposes of this chapter is to determine how characteristics inherent to a workload affect its potential to benefit from techniques that exploit affinity. In later sections, we will look at the ability of different techniques to fulfill this potential. Section 3.3 studied the effect of cache data loss on the performance of multiprogrammed workloads using a standard UNIX scheduler. In this section we will consider how much of this loss can be recovered through cache affinity scheduling. We will first look at the characteristics of a workload's constituent applications that affect its ability to benefit from affinity scheduling, and then at how the interactions between applications in a workload affect the benefit.

4.1.1 Application Characteristics

There are three main characteristics that determine the potential for a process to exploit cache affinity: (i) the amount of cache data (or state) it reuses, (ii) the length of time it executes continuously without releasing the processor (the dispatch interval), and (iii) the reason for eventually releasing the processor. We now discuss these characteristics and their effect on gains from affinity.

Amount of Reused State and Duration of Dispatch Interval

As discussed in Section 3.3, the amount of cache data an application reuses and the length of that application's dispatch intervals control the performance losses it may suffer under standard scheduling, and thus the performance gains it may achieve under affinity scheduling. Since the goal of cache affinity is to reduce the amount of reused data that must be fetched into the cache more than once, an application with little cache reuse will gain little from affinity. Likewise, one with a large amount of data reused across dispatch intervals has the potential gain substantially from affinity. The length of the dispatch interval also affects potential benefits from affinity because it controls the proportion of time an application spends refetching data into the cache. If the dispatch interval is long, the time to load reused data into the cache will be small compared to the time spent in the steady state, and thus the loss of performance due to the extra misses will be small. Additionally, an application with little reused data per dispatch interval may gain from affinity if the dispatch intervals are sufficiently short.

The amount of reused cache state reuse and average dispatch interval lengths for the applications were shown in the previous Chapter in Table 3.1 and discussed in Section 3.3. Potential losses in the standard scheduler, assuming intervening processes completely replace available cache data, varied from 0–23%. Table 4.1 shows the exact data.

Table 4.1: Potential performance loss in standard UNIX scheduler from cache data loss assuming flushed caches.

Application	Loss (%)
`LocusRoute`	11
`Ocean`	22
`MP3D`	0
`Cholesky`	2
`Water`	10
`Pmake`	23

Reason Why Dispatch Interval Was Terminated

Even if a workload suffers substantial performance problems due to cache data loss with a standard scheduler, other factors may also limit the effectiveness of cache affinity scheduling. The third process characteristic that we consider is the reason why a dispatch interval was terminated. A dispatch interval may be terminated by the expiration of the time quantum, or by a number of events initiated by the process. These include blocking on a semaphore (usually due to I/O), unsuccessfully trying to acquire a lock, issuing a system call, suffering a TLB fault, and other less frequent events. Of the possible causes of termination, only semaphore blocks actually block the process. We distinguish between dispatch intervals that block the process and those that do not. In the former case, the process is less likely to become runnable in time to reuse its state left in the cache. Therefore, the potential for cache affinity is smaller.

The causes for dispatch interval termination are presented in Table 4.2. As expected, scientific applications rarely block. For `Water`, 90% of the dispatch intervals end due to time quantum expiration. `LocusRoute`, `Cholesky`, and `MP3D` have a high percentage of dispatch intervals interrupted due to failure to acquire a lock. The only application with a large amount of blocking is `Pmake`, which ends almost half its dispatch intervals by blocking.

Table 4.2: Breakdown of events that terminate dispatch intervals.

Cause	`LocusRoute` (%)	`Ocean` (%)	`MP3D` (%)	`Cholesky` (%)	`Water` (%)	`Pmake` (%)
End of Quantum	2.7	26.3	24.8	2.2	90.1	21.5
Semaphore Block	0.4	13.2	1.5	0.1	1.1	47.5
Synchronization	96.5	47.8	71.4	97.4	6.3	0.0
System Call	0.2	1.2	0.8	0.1	1.9	14.8
TLB Fault	0.1	6.8	0.7	0.0	0.0	10.3
Other	0.1	4.7	0.8	0.2	0.6	5.9

4.1.2 Workload Characteristics

When we run multiple applications concurrently as a workload, interactions among the applications can affect the potential benefit of exploiting cache affinity. There are two major interactions: the way dispatch intervals of different processes interleave, and the amount of cache state that this interleaving causes. We consider each in turn.

Effective Time Slice

The amount of time a process runs on a processor without intervening processes running is called an *effective time slice*. An effective time slice may contain several dispatch intervals if the same process is re-scheduled several dispatch intervals in a row. If this occurs frequently, there will be little potential for benefit from cache affinity since applications will already be exploiting cache state by running on the same CPU for a long time. This situation, however, is infrequent in traditional schedulers. To illustrate this, Figure 4.1 shows the distribution of the length of the effective time slices for our workloads using the standard UNIX scheduler on our machine. Although there are a few times when enough processes block that the remaining running processes can run for multiple time quanta, this is rare. The rest of the time, the effective time slice curves match the dispatch intervals of the constituent applications. This shows that there is still potential for cache affinity to be exploited in these workloads. (Note that the curves do not include effective time slices shorter than 1 ms because of their irrelevance to the overall performance of the workloads.)

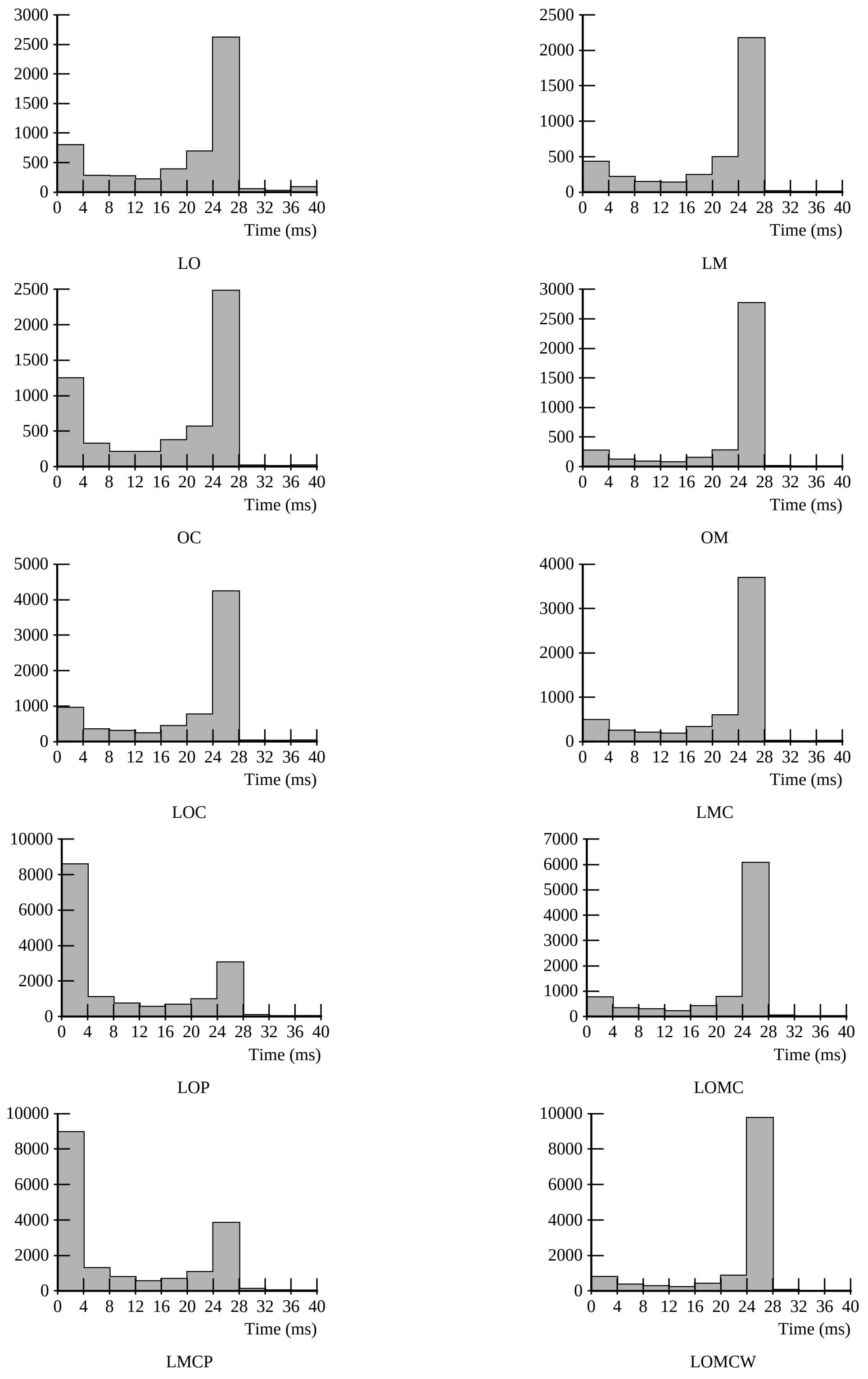

Figure 4.1: Distribution of the length of the effective time slices for the workloads studied. Effective time slices shorter than 1 ms are ignored.

Effect of Intervening Applications

When several processes share the same cache, they displace each other's state from the cache. This effect decreases data reuse in the workload and therefore increases the potential of cache affinity. We approximately estimate the amount of cache displacement caused by an application by the total number of misses it suffers in an average dispatch interval. Table 4.3 shows the number of times each application misses in a dispatch interval. `Water` suffers as many as 14000 misses in a dispatch interval, effectively replacing the second-level data cache. `Ocean` similarly suffers close to 10000 misses. `MP3D` causes about 3000 misses, wiping a large part of the first-level data cache. The other workloads have a smaller effect because their dispatch intervals are smaller.

Table 4.3: Number of misses per dispatch interval.

Application	Misses (Thousands)
`LocusRoute`	0.5
`Ocean`	10
`MP3D`	3
`Cholesky`	0.5
`Water`	14
`Pmake`	1.5

4.1.3 Summary of Application and Workload Characteristics

With the above data, we can get an accurate picture of what workloads can benefit from techniques that exploit cache affinity. The most promising workloads are those whose processes (1) are costly to reload because of the large amount of reused data, (2) execute for short effective time slices, (3) block infrequently or not at all, and (4) are interleaved with other processes that replace a large part of the cache when executed.

In summary, we believe that due to the large number of factors that can limit the potential benefits of an application for exploiting cache affinity, few applications are able to benefit greatly.

4.2 Implementation of Affinity Scheduling

To achieve the potential for benefit from affinity described in Section 4.1, we need an implementation of affinity scheduling that has low overhead and does not raise the possibility of starvation or loss of response time. A key goal of this research is to implement new scheduling approaches with minimal changes to the existing system, retaining fair allocation of processor time and good response time while increasing the performance of parallel applications. Our method of implementing cache affinity scheduling is to modify the existing process priority scheme of the

Table 4.4: Summary of process and workload characteristics that determine the potential of exploiting cache affinity.

Workload	Cache Reload Misses (Thousands)	Median Eff. Time Slice (ms)	Frequent Process Block?	Misses of Intervening Application (Thousands)
LO	0.2+2.2	5+15	N+N	10+0.5
LM	0.2+0	5+25	N+N	3+0.5
OC	2.2+0	15+5	N+N	0.5+10
OM	2.2+0	15+25	N+N	3+10
LOC	0.2+2.2+0	5+15+5	N+N+N	10.5+1+10.5
LMC	0.2+0+0	5+25+5	N+N+N	3.5+1+3.5
LOP	0.2+2.2+0.5	5+15+2	N+N+Y	11.5+2+10.5
LMCP	0.2+0+0+0.5	5+25+5+2	N+N+N+Y	5+2.5+5+4
LOMC	0.2+2.2+0+0	5+15+25+5	N+N+N+N	13.5+4+11+13.5
LOMCW	0.2+2.2+0+0+2.5	5+15+25+5+30	N+N+N+N+N	27+18+24+27+13

standard UNIX scheduler. We first describe the existing system, then discuss the modifications required to add affinity scheduling and their effect.

The standard UNIX scheduling system orders processes based on CPU utilization and run time. Runnable processes are placed in a run queue, ordered by a single number denoting their scheduling priority. A processor selecting a process to run picks the highest priority process from the queue. The priority is based in large part on the past CPU usage of the process it is associated with. As a process accumulates CPU time, its priority is reduced. This gives processes that frequently block more chances to run. The accumulated CPU time is periodically decayed to a fraction of its former value, however, so that long-running processes are not completely starved in favor of newer processes.

The exact algorithm is as follows. The priority of a process is denoted by its inverse, inv_prio, defined to be

$$inv\_prio = base + \frac{cpu}{2}.$$

A low value of inv_prio designates a high priority, that is, a process that will be chosen earlier to run. $base$ can be assumed to be equal for all processes we will be considering. cpu is a measure of the CPU utilization of the process; it is initially 0 for a new process and is incremented whenever a timing interrupt (raised every 10 ms on our machine) is received while the process is running. Finally, the value of cpu is decayed periodically to put a limit on the priority and avoid starving long-running processes.

To add affinity to the existing system, we temporarily raise the priorities of processes that are "attractive" from the standpoint of affinity scheduling when searching the run queue. We subtract a constant factor, a_p (for *processor*), from the inv_prio values of processes whose most recent execution was on the processor that is scheduling. This discourages migration between processors. We subtract another constant, a_t (for *time*), from the in_prio value of the process that has just finished executing on the scheduling processor. This encourages processes to run for consecutive dispatch intervals. Both adjustments are just for the purpose of scheduling at that moment, and the priorities relapse to their normal values after the processor has selected a process to run.

Given the priority algorithm, we can compute the effect of a given value of the constants. In our system, a workload using affinity has a maximum effective time slice of roughly $40a_t$ ms. The system restricts a process from migrating unless it receives $20a_p$ less processor time than other processes within a decay interval.

The modification is successful in a number of ways. First, it requires only minor modifications to the existing scheduler. Second, it is very efficient, since it only involves priority comparisons between at most three processes. In fact, given minor changes to the run queue structure, we can examine these processes without searching the run queue. Finally, there is no risk of unfairness or starvation since the normal priority system is still in place. Processes that are not executed have their priorities increased with respect to running processes and eventually run when the priorities are high enough to overcome the affinity adjustments. New and I/O-bound applications normally have high enough priority to be able to run as soon as they become runnable.

4.3 Performance Results

We begin by presenting the base results, obtained by using affinity scheduling as described in Section 4.2, both with small changes to process priority and with larger changes. We then evaluate two alternative ways of exploiting cache affinity, namely attaching processes to processors and increasing the time quantum of the machine.

4.3.1 Base Results

This section discusses the performance of the affinity function described in Section 4.2. We study two different degrees of affinity scheduling. First, we consider a scheduler with light affinity (*LightAff*) by setting both a_t and a_p to 6. This corresponds to a potential effective time slice of about 240 ms or 8 time quanta, and a migration threshold of about 120 ms. Second, we study a scheduler with heavy affinity (*HeavyAff*) by setting both values to 16, for an effective time slice of about 640 ms and a migration threshold of about 320 ms.

Figure 4.2 compares the performance of the two affinity scheduling schemes to the performance of the standard scheduler. The figure shows the execution times of the workloads, normalized with respect to the execution times of the same workloads under standard UNIX scheduling.

Our observations on the effect of affinity are as follows:

- **Cache affinity scheduling speeds up scientific workloads slightly.**

  With *LightAff*, each of the workloads runs faster under cache affinity scheduling than it did under standard UNIX scheduling. The improvement varies from 0–10%, with a geometric

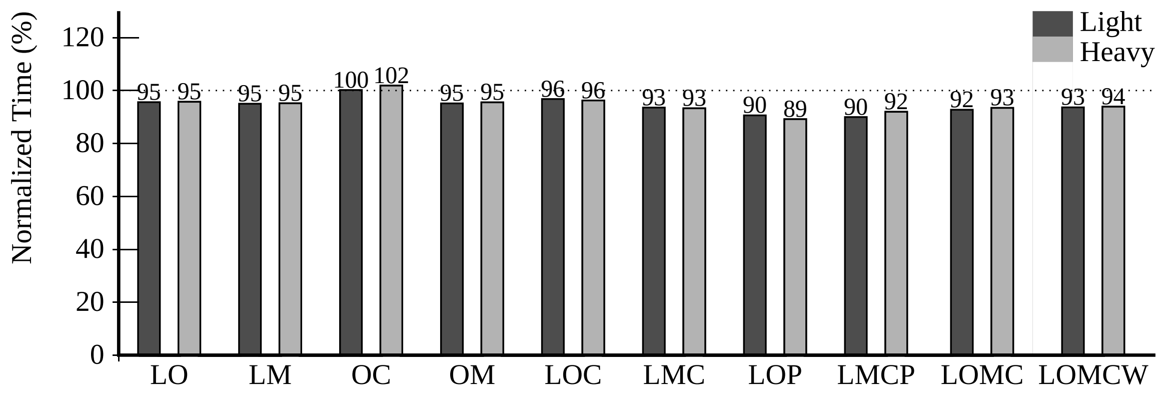

Figure 4.2: Performance of workloads under light and heavy affinity scheduling, normalized with respect to standard UNIX performance.

mean of 6%. The distribution of effective time slices with affinity scheduling is shown in Figure 4.3. In the figure, we see that the new effective slices are much longer than before, often more than 13 time quanta. There are still a large number of short effective time slices since these are the result of processes blocking rather than being preempted by the kernel, and are unaffected by cache affinity scheduling. The actual performance gains of affinity are small, because (as we saw in Chapter 3 and Section 4.1) a number of workload characteristics can limit the effect of affinity scheduling.

- **Cache affinity scheduling also speeds up mixed workloads of scientific and non-scientific applications.**

  As we discussed in Section 4.1, the `Pmake` application has good potential for gains from affinity. The application has a large amount of reused data and processes have short effective time slices because they often block on I/O, synchronize, or issue system calls. Since the processes mostly block before being preempted, affinity scheduling does not affect the effective slice noticeably. However, by encouraging processes to return to the CPUs on which they last ran, we can eliminate a number of misses. This is indicated by the performance of workloads LOP and LMCP, each of which improves by 10% with affinity scheduling, and which are the most improved workloads in the suite.

- **Implementation issues can affect the results.**

  The small changes necessary to support affinity scheduling often interact with other functions in the system. We briefly describe an example of this.

  When a process unsuccessfully tries to acquire a lock, it yields the CPU. The expectation is that the process that holds the lock will be scheduled. However, under affinity scheduling, the yielding process has its priority boosted by affinity and therefore is picked to run again. To avoid this, the process is prevented from being selected for this reschedule. While this solves the problem, it still allows the CPU to interleave between two processes trying to

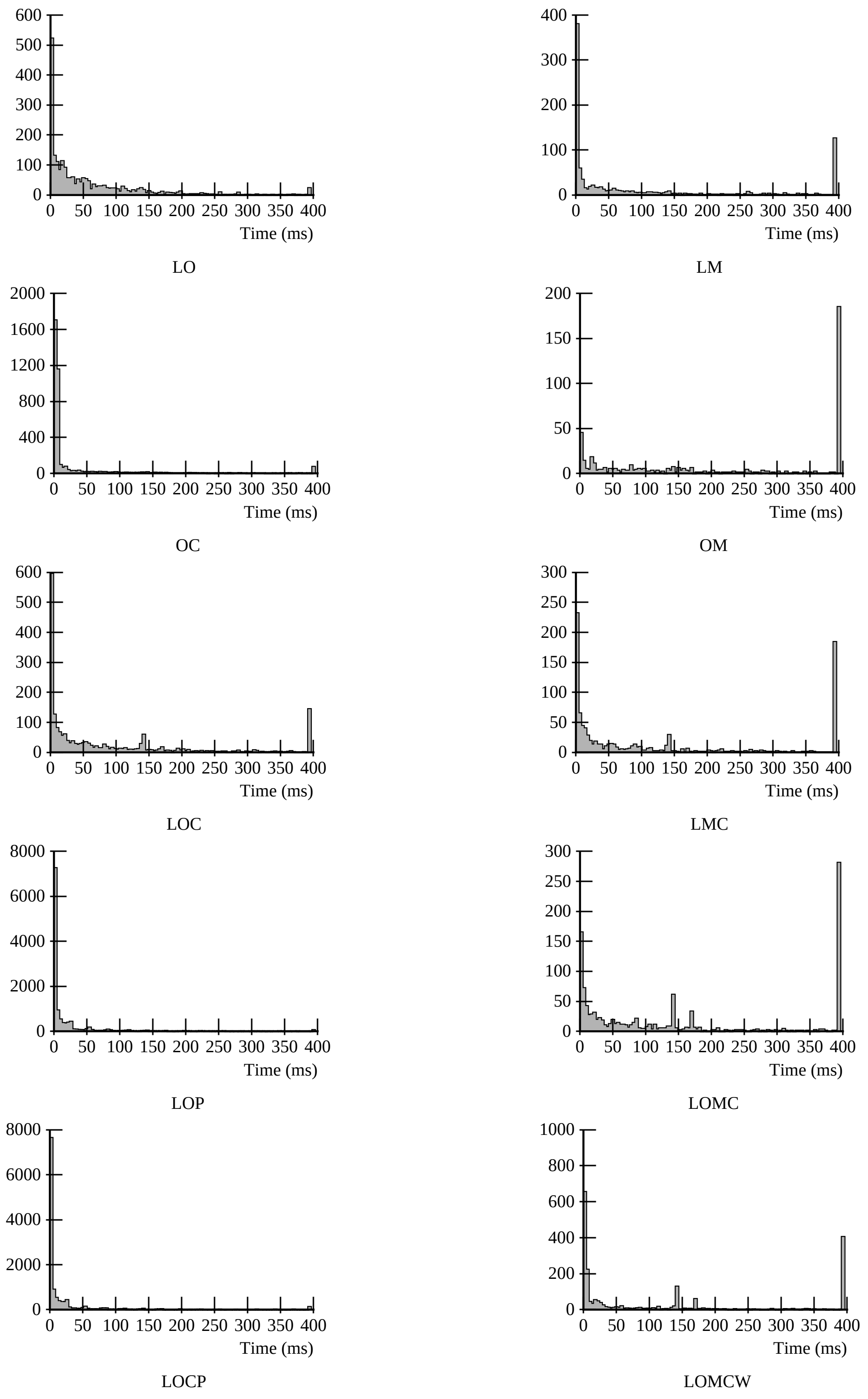

Figure 4.3: Distribution of the length of the effective time slices under affinity scheduling. Effective time slices lasting longer than 400 ms are shown as 400 ms.

acquire a lock. If the process holding the lock does not have affinity for that CPU, it may not be able to run until the CPU for which it has affinity becomes free. This effect can cause affinity to slightly slow down workloads with synchronization-intensive applications like `Cholesky`, as seen in the performance gain of only 0.1% for workload OC.

- **Increasing the level of affinity to large values does not improve performance.**

  Increasing the level of affinity from *LightAff* to *HeavyAff* would seem to increases state reuse, thereby reducing the number of misses. However, the effect on execution times is minimal or even negative. In fact, only workloads LOC, LMC, and LOP perform better under *HeavyAff*, and the improvements are slight. We conclude that a low level of affinity is sufficient and desirable. Hence, for the remainder of this thesis, we will use *LightAff* as our affinity implementation, and refer to it as simply “affinity”.

Overall, we note that our affinity algorithm fulfills the potential described in Section 4.1 quite well, producing low to moderate speedups of up to 10%. In addition, affinity did not cause significant problems with load imbalance. We also note that affinity does not introduce unfairness. The problem of unfairness would appear if, while encouraging one process to reuse cache state, we were consistently denying another process a fair share of CPU time. Our implementation is not unfair because a favored process will eventually accumulate CPU time and have its priority reduced with respect to processes that are not running. We have observed that applications with similar CPU-using characteristics are given equal CPU time.

4.3.2 Attached Scheduling

A simple alternative to our affinity scheduling algorithm is to fix processes on CPUs for the processes’ entire lifetimes. This strategy tries to increase the reuse of cache state by eliminating process migration without changing the effective time slice. Figure 4.4 shows the performance of this strategy, compared with the performance with affinity scheduling. While attached scheduling works well for very regular applications like `Ocean` and `MP3D`, it has problems with a more heterogeneous environment. This is due to increased load imbalance in the machine. When a CPU has no processes to run, it is forced to remain idle even if another CPU has more than one process to run. The result is an average improvement of only 1.6% over standard UNIX scheduling, much lower than affinity scheduling. In summary, we find that attaching processes to CPUs is not an acceptable alternative to affinity scheduling.

4.3.3 Increasing the Time Quantum

The simplest way to increase the reuse of cache data is to use a standard scheduler but increase the nominal time quantum of the machine. This strategy is the converse of attached scheduling—it tries to extend the effective time slice of processes without limiting process migration.

Figure 4.5 shows the performance of the standard scheduler with time quanta of 100 milliseconds, compared to the performance of affinity scheduling. We see that the workloads all run slower with 100 ms time quanta than with affinity scheduling. In many cases, they run slower than with 30 ms time quanta and standard UNIX scheduling. If we look again at Table 4.2, we

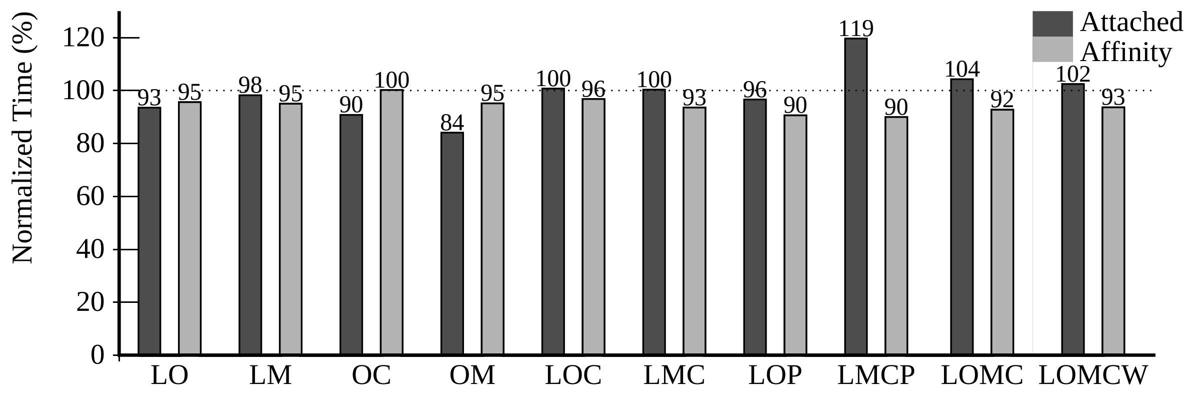

Figure 4.4: Performance of workloads under attached scheduling and affinity scheduling, normalized with respect to standard UNIX performance.

see the reason for this. For five of the six applications, less than 30% of the dispatch intervals are halted by time quantum expiration, with only a 30 ms time quantum. Increasing the time quantum does not help most of the applications. It additionally causes problems for blocking processes, since when a blocking process resumes it may be forced to wait longer to execute.

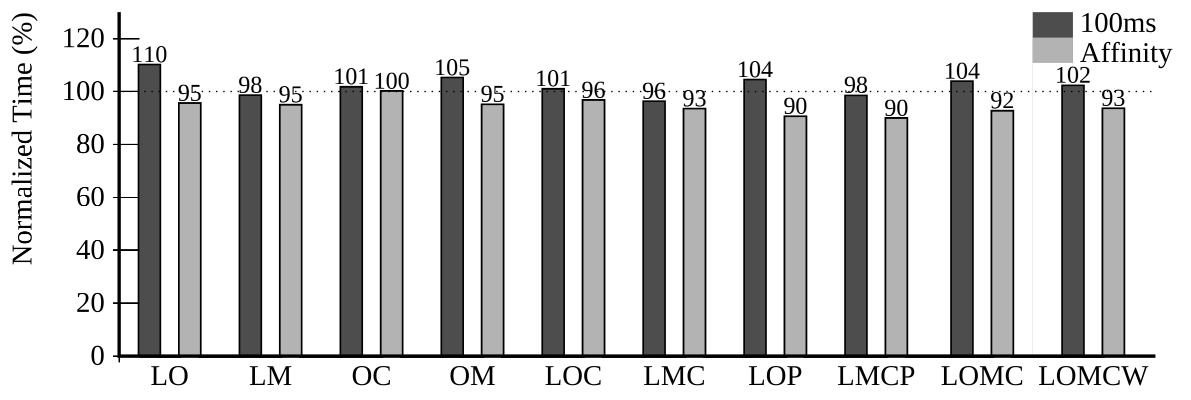

Figure 4.5: Performance of workloads with 100 millisecond time slices, and with affinity, normalized with respect to standard UNIX performance

4.4 Discussion and Related Work

The results we gathered are dependent on the nature of the machine being used. The Silicon Graphics machine has two data caches, a primary and a secondary, that are fairly close in size (64 Kbyte and 256 Kbyte) but not in speed (1 cycle and 14 cycles). This means that well-tuned applications will solely use the primary cache. In fact, all of our applications except `MP3D` and `Pmake` use the primary cache, and neither of those exceptions makes efficient use of the secondary cache. Although the primary cache is fairly large, it is small enough that refilling it every time quantum does not severely hurt performance.

The advantages of affinity scheduling would also be more pronounced on a machine with longer miss latencies. This is particularly true of the new generation of NUMA machines, where the miss latency varies and can be very long for remote misses. For example, on the cluster-based DASH machine developed at Stanford [24], a miss that is not filled by the secondary cache can take as long as 130 processor cycles. As the latency increases, the time taken to fill the cache increases, and thus affinity becomes more important. On these machines, issues of geographical affinity for a processor also come into play.

A number of other researchers have studied affinity scheduling using a variety of techniques and environments. However, few of these studies have made a detailed study of the benefits of affinity scheduling on a real multiprocessor with real applications. As a consequence, while their results are more broad-based, they do not consider many of the subtle issues and complex interactions of application, architecture, and scheduler discussed here. Squillante and Lazowska [41] measure response time under different affinity-based scheduling policies. Their results suggest that affinity scheduling provides substantial benefits. However, their results are based on analytical calculations with simple machine and application models, rather than a real implementation. Another analytical study was done by Squillante and Nelson [40] studying the effects of migrating processes between processors. They conclude that unconditionally fixing processes onto processors, while allowing processes to reuse more cache state, causes too much load imbalance which results in fairness problems and idle time.

Mogul and Borg [30] used address traces of a variety of real applications to study the potential for cache reuse. They found cache reload overheads of up to 8%, depending on the workload. In contrast to our studies, however, their study was confined to process switching on a single processor. Their conclusions were also based on the use of a fairly short 16 ms time quantum. Trace-driven simulations of real applications on a multiprocessor by the author along with Gupta and Urushibara [18] similarly point to small benefits of affinity, but again a small time quantum was used (10 ms). Also, I/O and other load variations were not considered. A study by Devarakonda and Mukherjee [10] evaluated the performance of a real implementation of cache affinity on a multiprocessor, but used synthetic workloads. Their results suggest that implementation issues and workload choice can have a large impact on the measured performance of affinity scheduling, but the synthetic nature of the workloads means that the complex facets of real applications were missed.

One study that did measure real applications on a real system, by Thakkar and Sweiger [42], looked only at the performance of a database system under an extreme form of affinity. They studied a database application with 12 to 24 processors, running with attached scheduling. They found a significant amount of cache state lost due to cache migration, and simply attaching processes to processors improved performance significantly. While our results disagree on the effectiveness of attached scheduling, the difference may be due to the larger number of processors they use, which cause problems with bus contention. The bus contention increases miss latency, increasing the potential for gains from affinity. Vaswani and Zahorjan [46] also used real applications and a real implementation of cache affinity. They found that due to the relative infrequency of process preemption, benefits were minimal. However, the results were based solely on scientific workloads and on a *space-sharing* scheduling system that partitions processors among applications, similarly to process control. The overall effect of the scheduler is to provide very long (over 300 ms) effective time slices even without affinity, so the small gains for affinity scheduling with scientific workloads are understandable.

4.5 Conclusions

In this chapter, we explored the benefits applications can achieve through the use of cache affinity scheduling. We found several characteristics inherent in applications that influence their potential for benefits: the amount of cache data an application reuses, the time its processes run continuously on a processor, and the frequency and duration of process blocking. We also found important characteristics of the ways applications interact when run concurrently in a workload: the effective time slice of their processes and the effect of other applications in the workload in removing data from the cache. We studied realistic workloads from a variety of domains and found that while moderate gains are possible, few have characteristics that allow for large gains from exploiting cache affinity.

We found that a simple and efficient modification to a standard UNIX scheduler, discouraging process migration and extending the effective time slice of processes, accomplishes affinity scheduling without load imbalance or fairness problems. The implementation improves the performance of our workloads by up to 10%, fulfilling most of their potential for benefiting from affinity. A small perturbation to the process priority was found to be sufficient (and even desirable) in attaining this performance.

We compared affinity scheduling to even simpler approaches for achieving this potential, namely permanently attaching processes to processor and extending the system time quantum. Although attached scheduling created too much load imbalance to be useful, extending the time quantum did work as well as affinity scheduling for most workloads. The exceptions are applications with processes that run only a short time before blocking and benefit from a reduction in process migration.

In the past, there has been some question as to the usefulness of providing affinity scheduling in an operating system. Based on the findings in this chapter, we conclude that affinity scheduling is a worthwhile addition to a standard scheduler. While few workloads are able to benefit substantially from exploiting cache affinity, affinity scheduling can be easily implemented, provides moderate gains for some applications, and does not hurt the performance of other workloads. In upcoming chapters, we will explore the effect of more radical changes to the scheduling system.

Chapter 5

Gang Scheduling

5.1 Introduction

In the previous chapter, we discussed trying to increase performance in multiprogrammed systems by scheduling using cache affinity. In this chapter, we will study another approach that tries to solve a related problem occurring in these systems. That problem is the synchronization inefficiencies that arise when processes of an application are not scheduled simultaneously. If a running process needs to communicate or otherwise synchronize with another process that is not also running, the delay involved may be substantial. Even when the applications use blocking or two-phase synchronization, the direct and indirect overhead from the increased context-switch rate (caused by increased blocking) can be substantial. The *gang scheduling* approach tries to avoid these costs by increasing the likelihood that if one process from an application is scheduled, all other processes from that application are also scheduled at the same time.

Chapter 3 found that little performance loss can directly attributed to synchronization problems when using applications with two-phase locks. However still, we study gang scheduling for several reasons. First, gang scheduling has been proposed by others as a solution to multiprogramming problems. We wish to compare the performance of gang scheduling to that of other scheduling policies on an equal basis, using the same machine and set of workloads. Second, we wish to perform a thorough evaluation of gang scheduling across a variety of multiprogrammed workloads; previous studies have done very little evaluation. Finally, although our implementation uses an approach similar to a prior implementation, we consider additional issues not considered in the prior work. As in Chapter 4, we wish to minimize changes to the standard UNIX scheduler, and preserve fair allocation of processor time and reasonable response time.

5.2 Previous Approaches

Gang scheduling was first proposed by Ousterhout as part of the Medusa project at Carnegie-Mellon University [31]. Ousterhout proposed several means of scheduling processes to achieve the desired goal. These approaches were later expanded on and used in several other systems [13, 14, 8, 35]. We will first discuss these approaches, then describe our approach to the problem and how it differs from the previous work.

5.2.1 Medusa

The basis for gang scheduling is the scheduling system proposed by Ousterhout for the Medusa operating system used in the Cm* project at Carnegie-Mellon University. Ousterhout suggested that one performance problem on parallel machines is that processes often form cooperative groups, and a running process often wishes to communicate with another in its group that may not be running. If the scheduler could be forced to schedule all processes from the same group simultaneously, this problem would become much more rare. This form of scheduling was called *coscheduling*. (“Coscheduling” and “gang scheduling” are often used interchangeably in the literature. This thesis uses gang scheduling as that seems to be the preference of more recent authors. This should not be confused the use of gang scheduling by other authors [4] to refer to dedicating a “gang” of processors to a single application until that application completes.)

Ousterhout proposed several methods for implementing gang scheduling on a multiprogrammed system. The simplest was the *matrix* method, wherein processes are scheduled based on their placement in a matrix, with rows representing time slices and columns representing processors. The scheduling algorithm places a group of processes on the first row in which they fit. The system then does round-robin scheduling through the rows of the matrix, with each processor scheduling the process in the column that corresponds to it. If a process blocks before the end of the time slice, or if a matrix entry is unfilled when its row is scheduled, the next process in the same column is scheduled. An arbitrary process cannot be selected since Medusa does not allow processes to migrate between processors once they have been scheduled.

Ousterhout referred to the proportion of time a scheduler spends running all processes from a parallel application simultaneously as the scheduler’s *coscheduling effectiveness*. As the coscheduling effectiveness drops, so too do the potential benefits from gang scheduling. One problem with the matrix method is internal fragmentation. If the number of processes used by applications does not always match the number of processors, there will often be rows that are only partially filled by a single application. Unless another application is later created that fits in the remaining portion of the row, many matrix entries will be empty. While processors can always search for another process to run when a matrix entry is entry, the coscheduling effectiveness is reduced and load balancing problems are more likely to occur.

Ousterhout proposed two other methods for combating this problem, the *continuous* method and the *undivided* method. The continuous method tries to avoid internal fragmentation by using a continuous array of processor scheduling slots, rather than a matrix with rows and columns. An application’s processes are then placed in the first set of available slots that could be enclosed within a single “window” as wide as the number of processors in the system. The kernel schedules processes by successively sliding this scheduling window to the beginning of the next application that has not had all of its processes run. Processors schedule the processes within the current scheduling window. The undivided method is similar, but only places an application into contiguous scheduling slots. Both of these approaches reduce internal fragmentation, but have problems with external fragmentation and did not perform better in simulated experiments. The simplicity and effectiveness of the matrix method have made it the most commonly used approach, both in Medusa and in modified forms in later systems.

5.2.2 Psyche

The Psyche project at the University of Rochester also looked into implementing gang scheduling [8, 9]. The Psyche implementation was based on Ousterhout's matrix method, but made several changes to make it more effective on real machines. The foremost variation was the method for coercing processors to select processes from the same application. Instead of forcing the scheduler to explicitly select process from a matrix row, as in Medusa, the Psyche system simply raises the priority of the processes in the appropriate row. Each processor then automatically schedules the gang-scheduled process on its queue. (Psyche has per-processor queues since the costs of migrating processes on the underlying machine, the BBN Butterfly, are high.) This has the advantage that if important system processes are waiting to run, they are scheduled in favor of a gang-scheduled application process. Also, if a gang-scheduled process blocks, the scheduler automatically schedules the next-highest priority runnable process on the local queue. This approach also involves no changes to the low-level process scheduler.

5.2.3 Other Approaches

The Silicon Graphics IRIX operating system [35] used on the machine used for experiments in this thesis provides another, simpler, approach to gang scheduling. Under IRIX, applications can either be gang-scheduled or non-gang-scheduled. Whenever a processor picks a process from a gang-scheduled application, it sends an interrupt to all other processors indicating that they should also start running processes from this application. A processor responding to such an interrupt attempts to schedule such a process if several conditions are satisfied (for example, if the current process that is scheduled on it has been running for at least 20 ms). While this allows gang-scheduled and non-gang-scheduled applications to coexist, the coscheduling effectiveness is not very good due to the frequency at which interrupts are generated. Whenever a gang-scheduling process blocks and a process from another gang-scheduling application is scheduled, interrupts are sent out that may cause some processes to be preempted while others are not. Also, the excessive number of interrupts can add substantial overhead to the system. We will look further into the performance of the SGI implementation in Section 5.4.

Feitelson and Rudolph [14] also investigated gang scheduling, in a somewhat different environment. They were interested in using gang scheduling in a hierarchical fashion on distributed machines. They point out the possibility of contention with a centralized gang scheduling system, and organize their gang scheduler in a distributed fashion. The machine is conceptually structured into a binary tree, where the leaves represent processors or clusters of processors. Each interior node of the tree contains a process scheduler. An application is placed in the smallest subtree in which it fits; that is, where there are enough processors to run all of the application's processes simultaneously. Multiple applications within the same subtree are then gang-scheduled. The idea is to organize the hierarchy to reflect the communication network of the system; processors that may communicate quickly can be grouped into the same subtrees. Although this is an interesting approach to gang scheduling for a distributed system, it would not be effective on a closely-coupled system such as the SGI. Since the environment and constraints of their system are substantially different than those of our system, we will not investigate their approach further.

5.3 Proposed Design and Implementation

Our solution to the problem of gang scheduling is similar to the Psyche approach. As in Medusa and Psyche, the processes in the system are placed into a matrix, where rows of the matrix correspond to time slices. Like Psyche, rows of processes are raised in priority to be gang-scheduled, but since a global queue is used the processes are selected in order of their relative priority. Unlike the earlier systems, though, the column a process is placed in has no significance; any process from a given row may run on any scheduling processor. Although this results in additional process migration, we saw in Chapter 4 that this has little effect on performance. This form of gang scheduling could also be combined with the affinity scheduling described in Chapter 4 to minimize process migration without load balancing problems.

Another difference between our system and Psyche (and Medusa) is that no explicit information from the application is needed. In Psyche, a parallel application must inform the kernel of the number of processes it will be using. The kernel then reserves a corresponding number of slots in a row of the scheduling matrix. We avoid the necessity of modifying applications by automatically adjusting as an application that was previously thought to be serial indicates that it is parallel by spawning processes.

Our scheduler works as follows. We maintain a two-dimensional linked list of applications. Each application contains one or more processes, and the number of processes can change dynamically. Each “row” of the linked list represents applications whose processes will have their priorities raised for a period of time. The system then reduces the priority of those processes and applications to the values their original values (plus or minus any priority changes that would have normally occurred during that time period), and raises the priorities of the next row. Each row contains one or more applications whose total number of runnable processes does not exceed the number of processors in the system. Although normally there will only be one parallel application per row, there may be more than one if each uses only a small number of processes, or serial applications may be combined with parallel applications that do not fill the system. We call this data structure a “matrix” to match Ousterhout’s terminology, even though the number of rows is variable and the “columns” are organized by application, not by processor.

When an application is created, it is assumed to be serial and is placed wherever there is an empty space in the matrix (where there is a row that is not yet full). If the application is in fact parallel, it will create new processes. When the row an application was placed in becomes too full due to new processes, it is moved to a newly created row. We create a new row rather than trying to find an existing row with enough room since we want to avoid moving the application too much (making a succession of moves as each row becomes too full), and we assume most parallel applications will use most or all of the system. In larger systems, this may not be the case and we may wish to vary the scheduling policy accordingly. When an application finishes executing, its processes are removed from the matrix, and its row is removed if it is now empty.

As Ousterhout found with the Medusa system, the matrix may become fragmented into many unfilled rows as applications of different sizes enter and leave the system. Although this does not result in idle processors, since empty space on a row simply means some processors run processes that are not currently coscheduled, this does reduce the effectiveness of gang scheduling by reducing the amount of time applications spend coscheduled. To solve this problem, we periodically *compact* the matrix, moving multiple applications onto the same row whenever possible. We do this by creating a new, empty matrix and placing applications in the new matrix in order of their number of processes, largest first. We then switch to using the new matrix in favor of the old one.

We think that this approach provides a simple yet effective approach to gang scheduling. The scheduling system requires no intervention from the user level and thus no modifications to existing applications. The scheduler coschedules parallel applications while still scheduling processor time for serial applications. While response time for interactive and I/O-bound applications may suffer slightly while coscheduled applications are executed, they are explicitly given processor time in the matrix, and even preempt gang-scheduled compute-bound processes if their priorities are sufficiently high (as is typically the case with intermittently executing processes). The compaction algorithm provides a necessary adjustment of the matrix, periodically moving applications to maximize coscheduling.

5.4 Performance Analysis

To measure the performance of gang scheduling for multiprogrammed applications, we measured the execution time of the suite of workloads. We used two forms of gang scheduling: the standard SGI implementation, and our method described in Section 5.3 (referred to henceforth as the "Stanford" implementation). Figure 5.1 shows the performance of the workloads with both gang schedulers, normalized with respect to the performance with the standard UNIX scheduler.

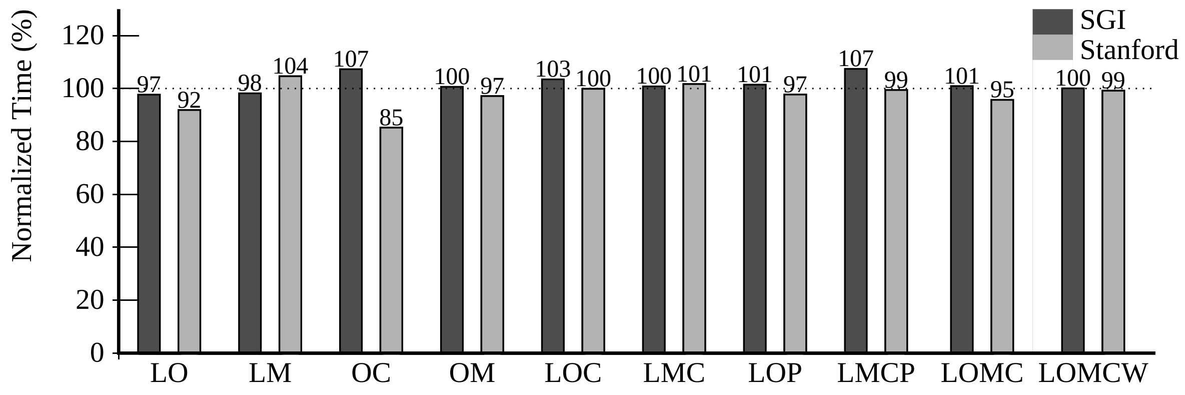

Figure 5.1: Performance of workloads under the SGI and Stanford implementations of gang scheduling, normalized with respect to standard UNIX performance.

We can make several observations from this data. First, we see that the SGI implementation of gang scheduling has poor performance. In only three of workloads is the performance better than that of the standard UNIX scheduler; in the rest the SGI gang scheduler is worse. This is because of the excessive interrupts discussed in Section 5.2.3. We found that when three gang-scheduled applications were run simultaneously, the coscheduling effectiveness of the SGI implementation was only 12%. Moreover, gang-scheduling interrupts were generated approximately once every 5 ms. Given that the cost of sending and receiving an interrupt is approximately 100 μs, this implies an overhead of 2%. The low effectiveness combined with the interrupt overhead made the SGI gang scheduler perform worse than even the standard scheduler.

The Stanford gang scheduler, on the other hand, is much more effective. Figure 5.2 shows the coscheduling percentage for each workload, both for a normal UNIX scheduler and for the gang scheduler. As the table shows, the standard scheduler results in very low values, all under 10%. On the other hand, the gang scheduler is very effective when only parallel compute-bound applications are running, with percentages in the 70–90% range. Gang scheduling is more difficult for workloads that include `Pmake`; while full coscheduling is much more common than with a standard scheduler, the percentages are still only about 40–50%. The I/O-intensive nature of the `Pmake` processes tends to disrupt gang scheduling, since I/O-bound processes often have high enough priority to interrupt priority-boosted (coscheduled) compute-bound processes. The Stanford gang scheduling implementation also has much lower direct overhead than the SGI implementation, since interrupts are be sent out only when the "row" being coscheduled changes. In our implementation this was every 30 ms, and could be set to much longer. The resulting overhead, with the same assumptions about interrupt cost, is only 0.3%.

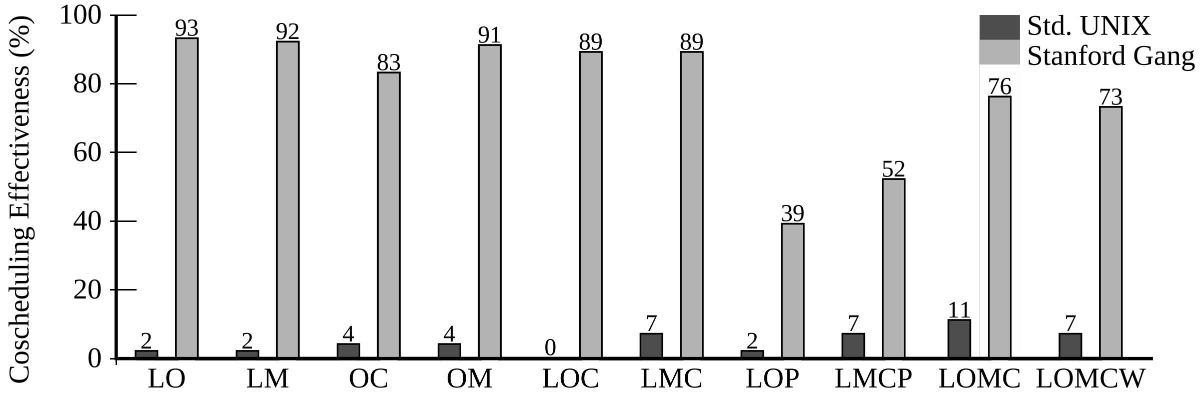

Figure 5.2: Coscheduling percentage for Stanford gang scheduler as compared with standard UNIX scheduler.

While the coscheduling percentages of the workloads are vastly increased with gang scheduling, we see that the performance is improved only slightly. The best performance was achieved by workload OC, which were sped up by 15%. The improvement of the other workloads was below 10%, with a couple even performing slightly worse with gang scheduling. Average improvement, using the geometric mean of the workloads, was only 3.5%. By looking at the performance of individual applications within the workloads, we have found that the execution times of `LocusRoute`, `Ocean`, and `Water` are improved by gang scheduling. `Cholesky`, `MP3D`, and `Pmake`, however, appear not to benefit (although they may execute faster in some workloads, this can be attributed to the increase in available CPU time when other applications in the workload complete executing more quickly). When the first three applications form the majority within a workload, the performance gains with gang scheduling are good (if small); when the latter three applications form the bulk of the workload, the gains are poor.

The poor performance of `Pmake` is expected, and can be attributed to a slightly slower response to I/O completion under gang scheduling. A process that has just become ready to execute due to I/O completion has to compete with processes with boosted priority (the currently coscheduled row) for processors. This could be easily addressed by tuning the priority system to decay priorities more slowly, giving I/O-bound jobs more weight. The result is better performance for the `Pmake` application and slightly poorer performance for simultaneously running CPU-bound applications.

The question remains of why gang scheduling does little for `Cholesky` and `MP3D`. The answer is similar to the reason they did not benefit greatly from affinity scheduling in the previous chapter. Both applications have little reuse of cached data across dispatch intervals. As we saw in Chapter 3, with two-phase locks, the main benefit of gang scheduling is reducing the context-switch rate. Since with these applications the cost to reload data into the cache after a context switch is low, the only benefit of a reduced context-switch rate is the negligible direct cost of context switches.

5.5 Conclusions

This chapter examined the benefits of using gang scheduling to address synchronization problems in multiprogrammed systems. A number of previous researchers have implemented gang scheduling systems. One such implementation, developed by Silicon Graphics, was tested and found to be ineffective at coscheduling parallel applications.

The Stanford implementation, introduced in this chapter, borrows from the original gang scheduling work by Ousterhout and later enhancements in the Psyche project. This approach adds the option of process migration to increase coscheduling effectiveness, and includes a more sophisticated policy for dealing with serial, interactive, and I/O-bound applications. The implementation was developed with the goal of increasing performance while retaining reasonable response time and fair allocation of processor time, while at the same time adding minimal complexity to the standard scheduler.

The performance of this gang scheduler was evaluated using the same workload set and base machine as the cache affinity scheduler in Chapter 4. The scheduler was found to be effective at coscheduling parallel applications with low overhead. However, the performance of workloads improved only slightly. Applications that were most sensitive to context-switching behavior improved the most. Applications that were not sensitive to context switching, including parallel applications with very small data sets, those with very large data sets that cannot fit in the cache, and serial applications, were unaffected. The result was an average performance improvement of 3.5% when using gang scheduling.

While this small improvement may not be worth the additional scheduling complexity required for gang scheduling, it should be noted that the applications used were not very amenable to improvements from gang scheduling. As pointed out in Chapter 3, applications using two-phase synchronization do not spend a large amount of time spinning waiting for a lock, regardless of the scheduling policy used. Improved synchronization performance simply reduces the number of times processes must yield the processor. Applications using solely spinning synchronization, particularly those not implemented using user-level task-queues (which often block a task and run another rather than spinning or blocking a process), should attain more benefits from gang scheduling.

Chapter 6

The Process Control Approach

The previous two chapters examined the merits of cache affinity scheduling and gang scheduling under a multiprogrammed workload. Both approaches involve only slight changes to the operating system kernel and no involvement at the application level. The modified schedulers generally provided small but positive improvements in the performance of applications. In the next two chapters, we will discuss and study a more radical approach that involves cooperation between the application and kernel.

In a basic sense, the concept of process control introduced in Chapter 1 is fairly simple. When an application is executed, a set of processors are assigned to it. The application is informed of the number of processors that have been assigned to it, and other applications from which the processors were taken are similarly informed of the loss of processors. All applications concerned (if they support process control) suspend or resume processes as appropriate so that the number of runnable processes matches the number of processors assigned to them.

While the process control approach is simple and intuitive at a conceptual level, its effective implementation requires addressing a large number of subtle issues. This chapter discusses some of these issues and describe the solutions we have adopted. It first discusses how applications may dynamically change the number of processes they are using. It then considers the problems associated with the implementation of process control in these applications, and the interaction necessary between the application and the operating system. It also discusses the policy decisions involved when partitioning processors among the applications, and techniques to ensure reasonable response time for interactive applications. It concludes with an example illustrating how the process control approach works on a real system, and a discussion of related work. The next chapter continues with an analysis of the performance of process control as compared to the standard UNIX scheduler and to the scheduling policies discussed in Chapters 4 and 5.

6.1 Application Programming Model Issues

Since the process control approach requires dynamic adjustment of the number of runnable processes in an application, a fundamental question that arises is what programming models allow this sort of dynamic adjustment. For example, a suitable programming model should be able to effectively utilize newly created processes, or previously suspended processes that are resumed at some point in time. Likewise, to support process control transparently to the applications programmer, the programming model should make it easy to determine when a process can be *safely suspended*, that is, suspended without potential starvation, loss of data, or significant loss of efficiency.

There are two ways in which the issue of safe suspension can be approached. Process suspension can either be *avoided* except when it may be clearly seen that it is safe, or the normal suspension can occur unhindered but a *recovery* procedure must be invoked if the suspension was performed at an unsafe point. The recovery procedure may consist of resuming the process until it reaches a safe suspension point, then suspending it again and continuing. The advantage of the avoidance-based approach is that it ensures correctness regardless of the synchronization mechanisms that are used in the application. The advantage of the recovery-based approach is that it generally allows processes to be suspended more promptly; on the other hand, it must be able to recognize the synchronization mechanisms used in case it suspends at an unsafe point. We chose the more conservative avoidance approach. Other work, to be discussed in Section 6.5.1, looked at the recovery approach, constraining synchronization to simple lock-unlock pairs.

Although the problem of determining safe suspension points is intractable for arbitrary parallel applications, the problem is fortunately simple for the large class of applications that use a *task-queue* model. In this model, applications are broken up into a number of tasks, and server processes repetitively select tasks from a queue and execute them. In these applications, a server process can safely suspend itself after it has finished executing a task and before it has selected another task to execute. A server process can also safely suspend itself before it has finished executing a task, as long as it places that task back on the task queue and it makes sure that the task is not inside a spin-lock controlled critical section. Furthermore, resumed or newly created processes can do useful work immediately by simply picking tasks from the task queue and executing them.

Since task-queue based models are widely used to implement parallel applications on shared-memory architectures, the applicability of process control is quite large. For example, one can find several programming languages with runtime systems based on the task-queue model [6, 16, 19, 22], and consequently all programs written in these languages follow the model. Similarly, all applications written using threads packages [1, 7, 11] follow this model, as do many independently written applications [17, 32, 39]. In these cases, process control can be added to the library or runtime system without changing application programs at all. Finally, the process control model can also be made to work for other programming paradigms that do not follow the task-queue model, though this requires extra support from the compiler or the programmer to identify safe suspension points. Appendix A describes the interface to a library that can be used to add process control to an application or runtime system with a minimum of changes.

Added complications arise in implementing process control for applications with varying amounts of parallelism. If an application cannot use all of the processors it is allocated, it should release the unused processors for the use of another application. In a task-queue based application, processors may be released via a system call when a process cannot find a task to run on the task queue. Processors may be recovered, via another system call, when new tasks are placed on the task queue. Like process suspension, this can be encapsulated within a library or runtime system, transparently to the application programmer.

For our implementation, we have added process control to the runtime system of COOL [6], a task-queue based object-oriented programming language. To ensure safe suspension, we suspend a server process only when its task has finished or when its task blocks on a blocking synchronization primitive.[1] Processes can be created or resumed asynchronously without restriction. Using the library routines in Appendix A.2, less than 100 lines of additional code were needed (mostly to handle the complexities of COOL's task-queue system).

6.2 Kernel-Application Interaction

Although applications using the task-queue or threads model allow the use of dynamic process control, the question still remains of how this control is to be implemented and how the responsibility of process control is to be distributed between the operating system and the application. In particular, the following issues need to be addressed:

- What system-level events should the applications be informed about to support process control?
- How are these events to be communicated to the application, given that they may occur asynchronously with respect to the application?
- Given the goals of process control, how should applications safely and efficiently respond to relevant events?
- How can the above mechanisms be implemented without incurring significant overheads and without radically changing currently prevalent operating system environments?

In this section we discuss the various tradeoffs that are involved in resolving these issues, and present the specific solutions that we have adopted in our implementation. Appendix A describes the exact interface implemented. Note that in the following discussion when we refer to an "application" implementing some facet of process control, we really mean the runtime system of the programming language or the threads package used by the application. In general, we expect all aspects of process control to be totally transparent to the applications programmer.

6.2.1 Identifying Relevant Events

In order to make use of process control, applications must be informed when events occur that may affect the number of processes they should be using. These events include kernel-controlled changes in the number of processors assigned to an application, as well as suspension and resumption of application processes on kernel semaphores. The latter includes blockages due to

[1]Note that the blocking synchronization primitives are at the level of the programming language and not at the kernel level. That is, when a task blocks, that task is put on a wait queue, and the server process picks another task from the task queue. It is not the case that the server process is blocked on some kernel queue and another process or kernel-level thread is run on that processor.

I/O transactions. As we will discuss in the next subsection, applications respond to these events by suspending and resuming processes as appropriate. First, though, since processing of events incurs costs, we need to determine which of the above events are truly relevant, i.e., require some action to be taken by the application to ensure good performance, and which are only marginally relevant and may be filtered away.

In general, the relevance of events depends on (i) the nature of the application (some applications may be better able to tolerate excess processes than others), (ii) the nature of the parallel machine (some machines may have higher penalties for poor cache hit rates than others), and (iii) the duration for which the mismatch between the number of processes and processors is expected to last. We consider events that cause a longer term mismatch to be more relevant than those that cause a short term mismatch. The tradeoff here is that if a process blocks only for a short duration, it may be better to let a processor remain idle than to resume another process in its place. The resumed process will probably suffer from poor processor utilization since its working set will not be in the cache, and it may also destroy the data cached by the blocked process. On the other hand, if the process blocks for a longer duration, the idle processor will result in lower performance than if a new process was created.

Even when the relevance of events is known, there still remains the question of where to filter out the events. The events may be filtered by the kernel before they are sent to the application, or they may be filtered by the application before any action is taken on those events. Significant flexibility is obtained by having the kernel communicate *all* events to the application, and having the application filter events. In this way, even if different applications (runtime systems of different programming languages) find different types of events relevant, they can all share the same kernel interface. The disadvantage of filtering at the application level is higher overhead. For example, if the application is informed about the asynchronous events via UNIX signals, the excess cost of sending and processing signals for irrelevant events may be non-trivial. The most efficient approach is to have the kernel filter system-level events, and only communicate those events that will be acted upon. This, however, forces the application to rely on the filtering provided by the kernel.

The approach we have chosen is as follows. The kernel always informs an application when the number of processors assigned to it changes, since these are expected to be long duration events. (As will be discussed in Section 6.3, changes are expected to happen primarily when new parallel applications enter the system or old ones finish.) However, for blockages on kernel semaphores, the kernel decides whether or not to communicate an event based on the duration for which that particular type of semaphore is expected to block. Events corresponding to short duration semaphores, such as reads from the disk buffer cache in memory, are filtered out by the kernel. Events corresponding to longer duration semaphores, such as actual disk accesses, are communicated to the application. Sometimes the duration for which a process blocks on a semaphore may be highly variable, for example, on semaphores associated with a UNIX pipe. In such cases, it would be best to send an event to the application only after some additional state regarding the semaphore has been checked. In our current implementation, however, we do not exploit this optimization, and the kernel communicates such events anyway. The application then does further filtering, particularly in the case where two events that are close together cancel each other's effects.

6.2.2 Communicating Events to Applications

Another important issue is how an application should be informed about relevant events; that is, how it is informed when a processor is taken away from it, or an extra processor is made available, or a process blocks on I/O.[2] The two factors that influence the choice of the mechanisms used are (i) response time, that is how quickly the application is made aware of relevant system-level events, and (ii) communication overhead, that is the computational cost of communicating the events. There are two main choices for mechanisms; the kernel could asynchronously inform applications when a relevant event occurs, or the applications could synchronously poll the kernel checking for relevant events.

The role of the kernel is obvious in providing quick response time to events, since the kernel knows exactly when processor assignments are changed, or when a process blocks on some kernel semaphore. It is not possible for application-based polling to be competitive in this regard [45]. The primary drawback of the kernel asynchronously signaling the application is the high overhead associated with signal handling. In contrast, polling can be made quite cheap by having a shared data area between the application and the operating system where relevant events are recorded. As a result, it appears that when response time is critical, kernel-based signaling would be the mechanism of choice, but when some slack can be tolerated, polling may be best to reduce overheads.

In determining the type of communication necessary for process control, we consider the conditions under which it is important that an application receive and respond to events quickly. Relevant events can be broken down into two types, those that indicate that the application should decrease its number of processes (suspend a process) and those that indicate that it should increase its number of processes (resume or create a process). In the former case, the cost of a delay is frequently small. Although it results in excess processes for a brief time, the processes can be simply scheduled onto the processors in a round-robin manner, and continue to do useful work. This is especially true for applications using two-phase rather than spinning synchronization primitives [18]. If the number of runnable processes is *less* than the number of processors, however, one or more processors will sit idle. Although this may be reasonable for very short periods to avoid excess context switching, any long idle time will result in performance loss.

Also, regardless of how quickly an event is communicated to the application, the application may have to delay in responding to it. This is particularly true of "suspend" events. The application must wait until a process reaches a safe suspension point before responding to the event by suspending a process. "Resume" events, however, can be acted upon immediately, by creating a new process or resuming a previously suspended one.

Based on the above considerations, in our implementation we chose to have the kernel signal an application when an event occurs that indicates the application should increase the number of processes it is using. The application can then immediately create a new process or resume a previously suspended one, as appropriate. When the application should decrease the number of processes it is using, however, we do not send a signal. Instead, each process polls the kernel

[2]Note that these events (aside from blocking on I/O) are asynchronous to the execution of the application. For example, the kernel may take a processor away from an application any time it considers appropriate, without regard to the current execution state of the application.

whenever it reaches a safe suspension point. Since polling in our implementation is much cheaper than sending and receiving signals, this reduces the cost of communication by almost one-half without affecting the response time of the application to such events.

At a more detailed level, the interaction between the kernel and the application takes place as follows. To communicate process control related information, the kernel and the application use a shared counter (one per application) that resides in the kernel address space. This integer counter is read-only for the application and reflects changes in the number of processes the application should have active (as per the process control philosophy). Whenever the kernel observes a relevant event, it suitably updates this counter. If the event was one that increased the value of the counter, indicating that the number of processes should be increased, the kernel sends a signal to the application indicating that action needs to be taken. On receiving the signal, or on reaching a safe suspension point, the application reads the kernel counter, and takes whatever action is necessary. The application bases its action on the change in the value of the kernel counter since it was last read by an application process. The application process simply compares the current value of the counter to the old value that had been recorded earlier, suspending processes if the current value is smaller than the old value and resuming or creating processes if the current value is larger.[3] It is not necessary for the application to modify the kernel counter, which avoids protection problems for the operating system.

The signal is sent by setting a bit in the process control block of one of the application's running processes. Since a process normally responds to a signal only when it is returning from a system call or context switch, we also send a special interrupt to the processor on which the chosen process is running, thus ensuring quick attention to that signal. There are several reasons why we use a shared counter instead of directly encoding event information in signals. First, the counter provides efficient polling of the necessary kernel information, allowing the previously described optimization of avoiding signals when the counter value is decreased. Second, the shared counter lets any process reading the counter, not only the one to which the signal was sent, respond to process control events. This flexibility helps improve response time. For example, consider the case when between the time the kernel decides to send a signal to a process and the time that it actually sends it, the destined process blocks. While the destined process can not respond to the signal until it unblocks, under our approach another process can take the necessary action. Third, the shared counter helps combine multiple process control events, reducing overhead. For example, if multiple processes need to be resumed, this can be determined simply by reading the kernel counter once and computing the difference between the old and the new values. Finally, we avoided using signals to encode information because under UNIX information may be lost when multiple signals are sent to the same process. As a result, in our implementation, the role of the signals is solely advisory. They are there to improve responsiveness, but they do not affect

[3]For the moment, we assume that applications can make use of any created or resumed processes. In Section 6.2.4 we will consider the problem of applications with limited parallelism.

the correctness of the process control algorithm.[4]

6.2.3 Effect of Delays on Performance

Let us now examine the effects on performance of the delays in our system. One cost is that a newly allocated processor may remain idle because the application has not yet resumed or created a process. We expect the waste of processor cycles due to this to be small. That is because (as discussed earlier) our implementation ensures that the signal handler is invoked quickly by sending an interrupt immediately after sending the signal in such cases. A second cost is that when a processor is taken away, until one of the application processes reaches a safe suspension point, we have more processes than processors. If this duration is small, say because the tasks are reasonably small, then the excess processes may not hurt performance as all processors can probably continue to do useful work. If the duration is long, however, the performance may suffer. One problem is that if a process is preempted while it is inside a spinlock-controlled critical section, the other processes may waste time spinning idly. In our implementation, we minimize such idle time by always using blocking synchronization primitives with a small amount of spin time before blocking. By making the spin time before blocking equal to the context switching time, we can ensure good performance for both short and long critical sections [18]. However, there still remains the disadvantage of worse cache behavior when there are excess processes. The negative impact of this factor is reduced in our implementation due to processor partitioning — since all processes within a partition are from the same application, they often have a significant amount of shared data.

We now examine some benefits of having a small delay between process control events and the corresponding actions. As was stated earlier, the duration for which a process blocks in the kernel is frequently unpredictable. Because of the unpredictability, process control signals are sent to applications for many short-duration blockages. In such cases, instead of immediately resuming a process in response to an initial blockage and then suspending one when the blocked process soon wakes up, it would be better if no action were taken. This would minimize the overheads of resumption and suspension and those of cache corruption. The implicit delays that are present in our system help performance in these cases. As an example, recall than when a process blocks in the kernel, the shared counter is incremented and a signal is sent to one of the other processes of that application to take action. If the blocked process resumes quickly, then by the time that the signal handler is run, the kernel may have already decremented the counter again. As a result, the signal handler will find the value of the kernel counter unchanged, and as desired, no action will be taken. Similarly, consider the case where an application has been asked to suspend a process. Now the application can not respond to this command until one of the processes reaches a safe suspension point. If during this time, the kernel counter is incremented

[4]A complication arises when the only running process of an application blocks on a kernel semaphore. As described so far, the kernel would find no other process belonging to that application that it can inform to create or resume a new process. For the duration of the blockage, all processes of the application are blocked and any processors assigned to the application are idle. To avoid this, the kernel temporarily wakes up the blocked process and send a signal to it indicating (via the shared counter) that the number of processes should be increased. After creating a new process or resuming a previously suspended one (if appropriate), the signal-handling process returns to its former blocked state.

(say because another process blocks on I/O or because another processor has become available), then the suspension command will be nullified.

6.2.4 Applications with Limited Parallelism

As discussed in Section 6.1, applications sometimes need to inform the kernel that they have been allocated more processors than they can use. In our implementation, a system call is available that informs the kernel that the processors allocated to the application should be limited to the indicated number. Alternatively, it may indicate that the limit should be set to a given number more or less than the current number of processors allocated. In either case, the new processor limit for the application is adhered to (fewer processors than the limit may be assigned, but not more) until the call is made again by the application. In our COOL implementation of process control, we reduce the maximum number of processes whenever a process cannot find work on the task queue. We increase the limit when more work is made available via task creation. As we shall discuss in Section 6.3.2, applications that limit processors in this manner are rewarded by being able to acquire more processors at a later time, when those processors are needed.

6.3 Processor Partitioning

To make the process control technique usable in realistic system environments, it is important to have some way of dividing, or *partitioning*, the processors in the system among the active applications. Processor partitioning allows process-controlled applications to be separated from non-controlled ones, avoiding problems with fair distribution of processing resources. Otherwise, the non-controlled applications may get an unfairly large fraction of the processing resources. Another benefit is that since processes running on any given processor are likely to be from the same parallel application (with common code and shared data), it helps to increase the cache hit rate, thus increasing processor utilization. Finally, processor partitioning helps avoid the bottleneck associated with a centralized scheduler with a single run-queue.

As just stated, the processor partitioning approach divides the processors in a multiprocessor among the applications needing service. This is to be contrasted with most scheduling strategies that time multiplex the processors among the applications. The basic construct in the processor partitioning approach is that of the *processor set*. Each processor set consists of a local run-queue and other related data structures. A high-level *policy module* is responsible for assigning both resources (processors) and tasks (application processes) to it. Each processor executes processes that have been assigned to its processor set in a regular time-sliced manner [4], though this can be changed on a per-partition basis. It is possible to have a processor set with no processors assigned to it, in which case the processes assigned to it will simply be waiting in the run queue.

The policy module plays a critical role in making processor partitioning effective. For example, it must decide when to create or delete processor sets, how to distribute the processors among the processor sets, and how to assign applications to processor sets. Furthermore, it must make these decisions in view of higher level goals. These in our case are to provide: (i) fast response time for high-priority I/O-bound and interactive applications, and (ii) high throughput for compute-intensive parallel applications. In the following paragraphs we discuss issues that arise in the design of processor sets and the policy module. Since the design space is very large, we use the details of our implementation to make the discussions concrete. Much experimentation, however, remains to be done in this area.

6.3.1 Grouping of Applications

We first explore the organization of processor sets, that is, how to associate applications with processor sets. One straightforward solution is to create a separate processor set for each application. This approach has two disadvantages. First, the number of processor sets may become very large, thus greatly increasing the complexity of processor allocation decisions. (Although the number of parallel applications running at any one time in a typical system may be small, the total number of applications is often much larger, especially if one includes serial applications such as compilers and editors and system processes such as network daemons.) Second, many of these applications may not be able to effectively use even a single processor for the duration it is allocated to them. An example would be a compiler process that performs a lot of I/O. As a result, making effective use of processors under this approach requires that processors be frequently moved between processor sets, which is both inefficient and makes the processor allocation algorithm difficult.

An alternative strategy is to create one processor set per *class* of applications. For example, we can have one processor set for all process-controlled applications, another for all non-process-controlled parallel applications, another for compute-intensive serial applications, another for OS daemons, and so on. As desired, such a strategy avoids the problem of non-controlled applications grabbing an unfair share of the processing resources. It also has the advantage that since processors are allocated in larger clumps, there is greater sharing of resources. For example, if one compiler process is not using a processor due to an I/O blockage, another compiler process in the same processor set could use it during that time. However, this strategy has the problems that the set of application classes is quite ad-hoc, and that processor allocation to these aggregate processor sets, with multiple applications each, is an extremely difficult task.

The approach that we have chosen is in between the above two. Instead of allocating a processor set for each application in the system, we only allocate processor sets for parallel applications. All other applications (e.g., network daemons, compilers, editors, etc) execute within a perpetual *default* processor set. When an application first begins execution, it always starts in the default processor set. Applications that establish themselves as being parallel are then migrated to separate processor sets. A processor set is deleted when all processes assigned to it have completed. The data structures, however, are saved and reused. If the number of processor sets begins to exceed the number of processors in the system, we assign multiple applications to the same processor set. Process control signals caused by changes in the number of processors allocated to a processor set are then sent to all applications in the processor set, so all applications have the same number of processes. An alternative would have been to continue to assign each parallel application a separate processor set (thus, some applications may have no processors assigned to them), letting the processor allocation algorithm ensure that all applications get a fair share of processors over some longer interval of time. This is also a reasonable option, but would have degraded response time.

6.3.2 Processor Allocation

The next major policy decision is the assignment of processors to processor sets. Our main goal is to fairly allocate processors among applications, but the existence of serial applications in the default processor set and parallel applications with varying levels of parallelism makes the problem more complicated. Our assumption in designing the policy module was that high loads in the default processor set are often transient, and that handling these loads with high priority is important to providing good response time. We also wish to reward parallel applications for limiting the number of processors they use. Finally, we wish to give short-duration applications slightly higher priority than long-duration applications, since they use up fewer overall resources.

We have devised a priority system for controlling processor allocation that addresses these issues. This scheme is loosely patterned after the process priority system in a standard UNIX scheduler (described in Chapter 4.2). Each processor set is assigned an *inverse priority*, inv_prio. A lower value denotes higher priority, and a higher value conversely denoted lower priority. This number is adjusted over time, based on the number of processors used by the processor set. A processor set's inv_prio is initially 0 when an application is first assigned to the processor set, and is adjusted whenever a new processor allocation is to be made, as follows:

$$inv\_prio = inv\_prio' + nt$$

where inv_prio' is the previous value of inv_prio, n is the mean number of processors that the processor set has used since the last allocation, and t is the time elapsed since the last allocation. At periodic intervals, the priority is decayed by multiplying it by c, a preset constant with a value between 0 and 1, as follows:

$$inv\_prio = inv\_prio' c.$$

Processors are allocated to processor sets in an iterative manner, using the inverse priorities of the processor sets as follows:

- When allocating a processor, the processor set with the lowest inverse priority is given the processor. If there is more than one with lowest inverse priority, one is chosen randomly.

- When a processor is allocated to a processor set, the inverse priority of that processor set is temporarily increased by a constant, s (for the duration of the allocation). That is, if a processor set has been allocated 3 processors so far in the allocation, its inverse priority is now $inv\_prio + 3s$ for the purposes of further allocation.

- If a processor set has been allocated all processors it is allowed (by a user-defined limit), it is removed from the pool of sets being considered for further processors. The allocation continues until all processors have been allocated (if all parallel processor sets reach limits, extra processors are allocated to the default processor set.)

Thus, given two sets, the difference in their inverse priorities indicates the difference in the number of processors each is allocated, assuming there are sufficient processors.

The values of c and s may be set to control the effect the "history" of a processor set has on its future allocations. With a high value of c, prior allocations are reflected in the (inverse) priority for a long period of time; with a low value, priorities decay quickly regardless of prior usage. With a high value of s, priority differences make only a small difference in actual processor allocation; with a low value, priorities have greater significance. After some experimentation, we chose the values $c = .95$ and $s = 20$ as values allowing reasonable behavior; applications are given some advantage for limiting their processor usage without completely skewing the overall goals of fairness. We also chose to decay priorities every 300ms.

There still remains the issue of how to treat the default processor set. The problem of determining how to fairly allocate processor time to both serial and parallel applications has not been resolved by the operating systems community. Since there may be more than one application in the default processor set, it has the right to higher priority than other processor sets containing only a single parallel application. However, those applications are serial applications and many will be I/O-bound or interactive, so the demands in terms of processor usage of the default processor set are normally low. Our solution is to treat the default processor set like other processor sets in processor allocation, except that the constant c used to set the priority is set to a lower value. This reduces the effect of processor usage on allocation with respect to the default processor set, effectively giving it higher priority. The number of processors allocated to the default processor set is also limited to the average number of processes on its run queue, unless no other processor set can use the additional processors. This avoids allocating processors to the default processor set that will sit idle due to a low process load.

Another policy question is the frequency with which processor reallocations must be done. More frequent processor reallocations would be better at adapting to varying application loads and at preserving fairness, but would add overhead as the costs of processor migration are incurred more often. In response to this, our policy module performs processor reallocations under two circumstances. First, whenever a new processor set is created or an old one is deleted (that is, when a new parallel application enters the system or an old one finishes) a processor reallocation is done. This means that reallocation always immediately adapts to application load changes. Second, to allow for adjustments even when applications are not entering or leaving the system, processor reallocations are done periodically. (This interval is currently 300 ms.) Processor set priorities are decayed at this time, in addition to the adjustments that are made whenever a reallocation is performed. In all reallocations, every effort is made to ensure that processors are not gratuitously moved between processor sets, as this destroys the data accumulated in processor caches by the applications.

As stated earlier, one of our goals is to provide fast response time to I/O-bound and interactive applications. In adhering to this goal, when the load in the default processor set is very low and sporadic, the policy as described above gives rise to the following tradeoff. If we keep one processor permanently assigned to the default processor set, then we guarantee good response time, but this processor will mostly be idle. It could probably have been better used by some other processor set. (This is especially important for machines with a small number of processors.) However, if we allow the policy module to take away all processors from the default processor set (for example, this would be done by the policy module if at the time of reallocation the default processor set had no runnable processes), then consider the situation of an interactive application that arrives soon after the last reallocation. It may have to wait until the next reallocation interval (up to 300 ms away) before getting any service, resulting in very poor response time.

The solution we have adopted works as follows. If at the time the reallocation is done the default processor set has no runnable processes, all processors are taken away from it. However, when a process is added to the previously empty run queue of the default processor set, a "partial" reallocation is performed, assigning a processor to the default processor set. This ensures that the response time is reasonable. (Note that the application from which the processor is taken away is informed of this, so that it may reduce the number of processes it is using.) When processors that are assigned to the default processor set become idle, they wait for a short interval of time (about 10–20 ms) for more work to arrive.[5] If they find no work, they return to one of the other processor sets and become usefully employed.

There is another practical complication that we need to address. In many machines, it is sometimes necessary that a process run on some specific processor. For example, on a Silicon Graphics 4D/340, the network driver does not protect global data, and thus any processes wishing to do network I/O are forced to run on processor 1. We address such situations as follows. When a process wants to run on a specific processor, the kernel places it on a special global queue with a note saying that it must be run on the requested processor. (This is the *must-run* queue in the Silicon Graphics IRIX operating system.) All processors check this global queue before their normal processor-set run queue, and execute relevant processes from this queue first. When such special operations are completed, the process is returned to its normal queue. Certain high-priority system processes may also be placed on the global queue for optimal response time. Since processes on the global queue run only for a short time on the processors that execute them, they should have little effect on the performance of applications assigned to processor sets.

6.4 A Process Control Example

We end this discussion of process control design and implementation with a simple example demonstrating process control and processor partitioning in action. Consider a four-processor parallel machine, with two CPU-bound parallel applications, A and B. Each of the applications may be adjusted via process control to use any number of processes, but initially starts with one process. The stages of execution are shown in Figure 6.1.

When application A begins, as in part (a) of the figure, it is the only application running in the system. It begins running in the default processor set, which has been assigned all 4 processors. The application then notifies the system that it is a parallel application. The processor partitioning policy module then assigns the application to its own processor set, and allocates 4 processors to that processor set. When this allocation has completed, the kernel sets the counter shared between the kernel and application A to 4 and sends a signal to A's process. When the signal is received, the process checks the shared counter, and creates 3 new processes. The process returns to performing work, and we now have 4 processes running on 4 processors, shown in part (b) of the figure, and a stable system.

[5]The reason for waiting for a short interval is the following. Consider a situation where a process is doing a lot of disk I/O. If every time it blocked on I/O we took its processor away, then it would not get the processor back for another 100 ms or so, even though the I/O may have completed much earlier. Waiting for a short period helps this situation considerably.

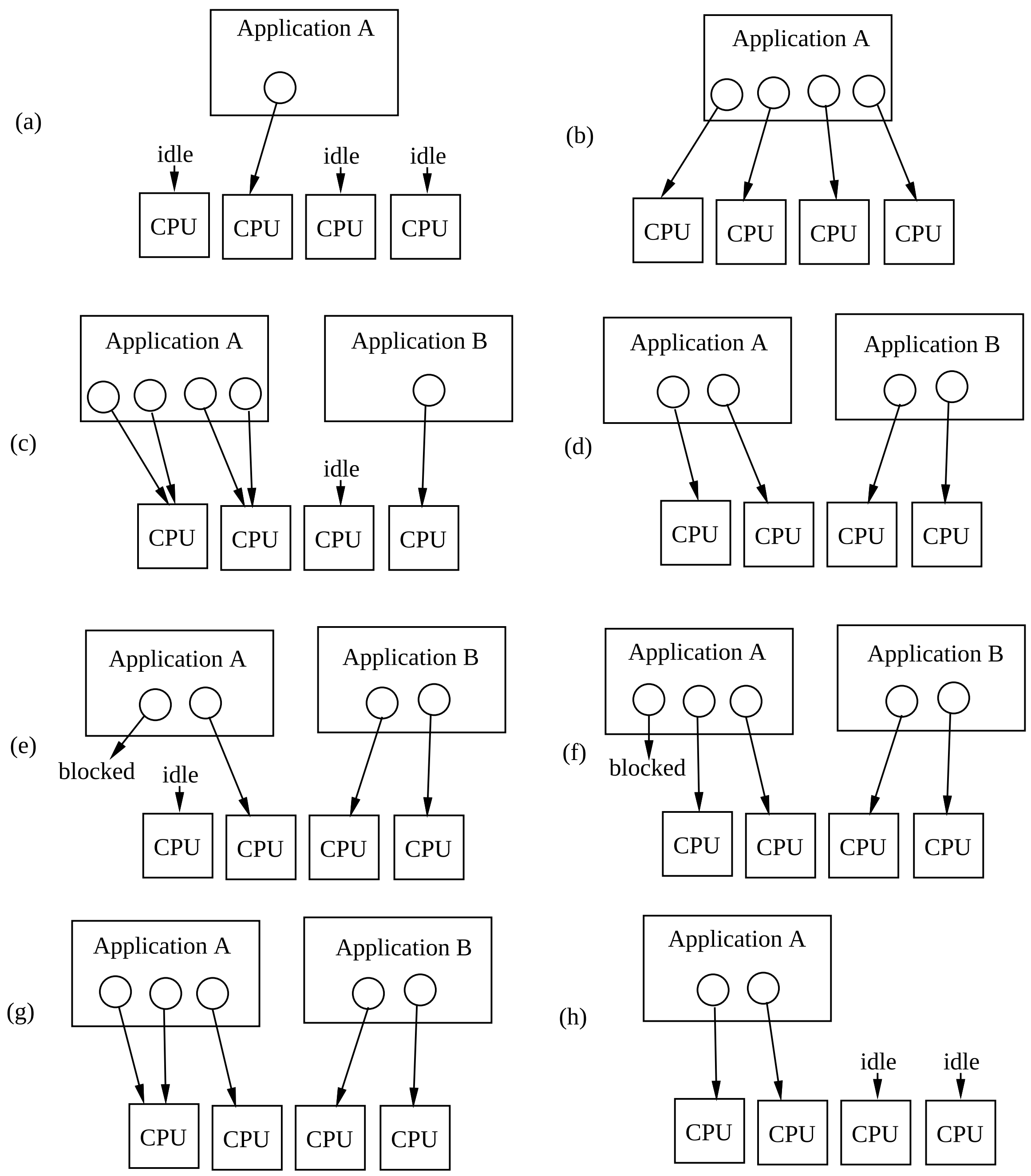

Figure 6.1: Stages of computation for a process control system with 4 processors and 2 applications.

Now, consider what happens when application B begins. Like A, B begins executing in the default processor set. The policy moves one processor to the default processor set to execute the application's initial process. This results in taking a processor from A. The kernel decrements A's shared counter by 1 (to 3). The first process of A's to reach a safe suspension point checks this counter and suspends itself. Application B notifies the system that it is a parallel application, and is assigned its own processor set. The policy module allocates B the processor previously allocated to the default processor set and one of the processors previously allocated to A. This stage is shown in part (c) of the figure. After this reallocation occurs, both A's shared counter and B's shared counter are set to 2 and a signal is sent to B's process. When B receives the signal, it checks the shared counter and creates a new process to take advantage of the extra processor. No signal is sent to A, but the first process of A's to reach a safe suspension point suspends itself (recall that processes always check the shared counter at safe suspension points). After this occurs, each application will have 2 processes executing on 2 processors, and we will have a stable system, as shown in part (d).

The next case we will consider is the effect of an I/O operation. Assume a process of application A does a read. The process is suspended pending completion of the read. This is shown in part (e) of the figure. When the process is suspended, the kernel increments A's shared counter (to 3) and sends a signal to one of the other processes of A. When that process receives the signal, it checks the counter and resumes one of its previously suspended processes. While the I/O operation is being performed, the extra process serves to avoid letting the processor that had been running the I/O-initiating process sit idle, as shown in part (f). When the I/O operation completes, the process that initiated the I/O is resumed. Before exiting the kernel, though, the kernel decrements A's shared counter (to 2). The process that initiated I/O returns to executing code. For a brief time, 3 processes are running on 2 processors. This stage is represented in part (g). The first process of A's to reach a safe suspension point suspends itself. When that process is suspended, the applications will again be at an equilibrium, as in part (d) of the figure.

Finally, consider what occurs when B completes before A. B's processor set is destroyed once application B exits and the two processors assigned to it are released and reassigned to application A. This is shown in part (h) of the figure. A's shared counter is set to 4 and a signal is sent to one of its processes. That process receives the signal and resumes 2 previously suspended processes. At the end of this, A will have 4 processes executing on 4 processors, as in part (b).

6.5 Related Work

Other researchers have also investigated the use of spatial partitioning as a way of handling multiprogrammed workloads. Scheduler activations is an approach with many similarities to process control that was developed in parallel with it, but has chosen different answers to some of our questions about design issues. Processor partitioning mechanisms have also been developed for use in allocating applications fixed numbers of processors for fixed periods of time.

6.5.1 Scheduler Activations

Anderson et al. at the University of Washington [1] have developed a system that is similar in concept and motivation to the process control and processor partitioning approach proposed in our previously published paper [45] and further developed in later work [18] and this thesis. In the Anderson work, *scheduler activations* are directly mapped onto processors assigned to an application in a one-to-one manner. (Scheduler activations are essentially the same as processes within an address space or kernel-level threads.) The scheduler activations are responsible for executing user-level tasks as per the task-queue model. When a task running on a scheduler activation blocks for I/O, or when the kernel assigns another processor to the application, the kernel directly and automatically creates a new scheduler activation. This scheduler activation is then run on the newly freed/allocated processor and it starts picking tasks from the task queue and executing them. Similarly, when the kernel wishes to take away a processor from an application, it directly intervenes and destroys one of the scheduler activations in a *safe* manner. Primitives are also provided for the application to inform the kernel when it cannot use the resources that have been allocated to it, or when it needs more resources. As we see, similar to the process control approach, there is a close correspondence maintained between the server processes and the number of processors allocated to an application. To the extent that the number of active processes matches the number of processors, both approaches eliminate the need for context switches and thus ensure low overheads and good cache performance.

While the approaches are conceptually similar, there are some differences in the underlying mechanisms and implementations. In the scheduler activations model, the user-level threads package and the kernel are more tightly coupled than in our approach. For example, the kernel directly creates and destroys scheduler activations in response to changes in numbers of processors, while in our approach the kernel only provides hints to the threads package which then resumes or suspends server processes. The method used to ensure safe suspension is also different in the two approaches. The scheduler activation approach uses a recovery-based method. To destroy an activation (when a processor is to be taken away), the kernel first preempts two activations saving their tasks. The kernel then creates a new activation which makes sure that the two saved tasks were not blocked within a critical section. It does so by further executing one of the saved tasks until it is outside of critical sections, then putting that task on the threads package's run queue, and then executing the second saved task. In contrast, we use a simple preventive approach in our implementation—we wait until an application process itself decides that it is at a safe point before suspending it. Although this introduces some delay in suspension of excess processes, our method does not require that all critical sections in the application code be clearly identified. In our experiments so far, we have not found the delays in suspension to be a problem. Rather, we have found that the slightly sluggish response of our approach helps prevent many short duration suspensions and resumptions of processes, which if done immediately would have degraded system performance. This would be even more true in large-scale multiprocessors [25] where the cost of cache misses can be very high, with each miss costing tens or hundreds of processor cycles. In such machines, it is often better to have the processor be idle for a few cycles, rather than bringing in a new task that may destroy the cache data accumulated by the previous task.

Finally, we note that the essential difference between process control and scheduler activations is the way in which signaled events are handled at the user level by the application and the accompanying runtime system. In our system, individual applications may use a scheduler-activations-like approach by simply making a system call with flags directing the kernel to send signals to the application on every process-control-related event, regardless of whether the action should be to suspend or resume a process. The runtime system could then employ an avoidance-based approach to process suspension, just as in scheduler activations, at the user level.

6.5.2 Processor Partitioning

The processor partitioning kernel interface we use owes much of its structure to the implementation of processor sets developed for Mach [4]. In Mach, users may flexibly create and destroy processor sets, assign processes to different processor sets, and allocate processors to processor sets (security restrictions notwithstanding). This allows a variety of scheduling policies to be implemented by user-level servers, though initially only a simple batch-based scheduling policy was implemented. In our system, we use the same basic mechanisms, but have implemented a sophisticated kernel-based policy module to control processor allocation. This lets us provide an integrated scheduling policy that provides processor partitioning and does not depend on user-level interaction and control.

In Section 6.1 we mentioned that applications need to be able to communicate their resource requirements to the kernel, so that processors that cannot be used by an application can be allocated elsewhere. The importance of using such information in scheduling decisions has previously been emphasized by Majumdar et al. [27] and Zahorjan and McCann [48]. In a recent report, McCann et al. [28] empirically evaluate the use of such a “dynamic” processor allocation approach and the results show noticeable benefits when running applications with highly variable parallelism. The benefits are relative to the performance of an “equi-partition” policy that is based on the concept of process control [45], but unfortunately lacks some important elements of our process control approach. The equi-partition policy described requires that a processor being moved between applications be “released” by the application to which it was allocated before the reallocation can take place. This results in problems with response time, as new applications must wait for processors to be released before executing, and in problems with fairness when running uncooperative applications (which do not release processors when requested). Our process control policy avoids these problems by allowing the kernel to reallocate processors as needed immediately, informing applications of reallocations but not waiting for the applications to respond.

6.6 Concluding Remarks

This chapter considered the design and implementation issues involved in implementing process control on a real system. We found that a wide class of applications can be implemented using a process control model. Process control and processor partitioning can readily be added to standard operating system kernels in an elegant and efficient manner. The result is a system that allows process control applications to run efficiently while still allowing other applications to run unaffected, time-slicing among available processors as usual.

Two major concerns in this work were the fair allocation of processor time to applications, and response time for interactive and I/O-bound applications. The *policy module* developed in Section 6.3 addresses these concerns by adapting the standard UNIX process priority system that handles these problems in a traditional system to a more complex *application* priority system. This policy module can also be easily adapted to system goals and constraints. For example, an environment that included animation would want to ensure very good response time, even at a slight cost to the performance of compute-bound applications. This can easily be done in our system by adjusting the parameters of the policy module.

In sum, the process control system described in this chapter seems to be a very reasonable solution to the performance problems raised by multiprogramming. The necessary changes to the kernel can be seamlessly integrated to create a system that looks like a traditional system except to appropriate applications. The next chapter will investigate the performance benefits that can be achieved using this system.

Chapter 7

Process Control Performance

Chapter 3 showed that as the number of processes executing an individual application increases, the efficiency of the application decreases. From this we found that we could improve performance substantially by controlling the number of processes based on the system load in process control style. The following sections present results for the system performance running real multiprogrammed workloads under process control schedulers. We first see that for our multiprogrammed workloads, the process control approach wins over other scheduling approaches by a significant margin. In the next section, we study the effect when both process controlled and non-controlled applications are present at the same time. We show that the use of the processor partitioning approach prevents the non-controlled applications from monopolizing the system resources, resulting in good performance for both kinds of applications. In the following section we present experiments that show the affect of our policy module on the response time of interactive applications. In the next section we show the effectiveness of the process control approach for applications performing a significant amount of I/O. Finally, in the last section we look at the effect of workloads where applications are executed at staggered intervals, so the number of applications in the system varies dramatically.

7.1 Multiprogramming Performance

In previous chapters, we considered adding affinity and gang scheduling to the traditional UNIX-style scheduling system. We will now investigate the effect of process control scheduling. We will also compare the results with an approach where processors are spatially partitioned among applications but the applications do not limit the number of processors they use (regular UNIX scheduling is used within each partition). Figure 7.1 shows the results when we run our standard workload set, both using only partitioned scheduling, and using process control with partitioning. As in previous chapters, the figure shows the mean execution time of the parallel part of the applications in each workload, normalized with respect to the mean execution times under the standard UNIX scheduler.

As the figures show, process control does significantly better than either the standard scheduler or using processor partitioning alone. If we compare with the cache affinity results in Chapter 4

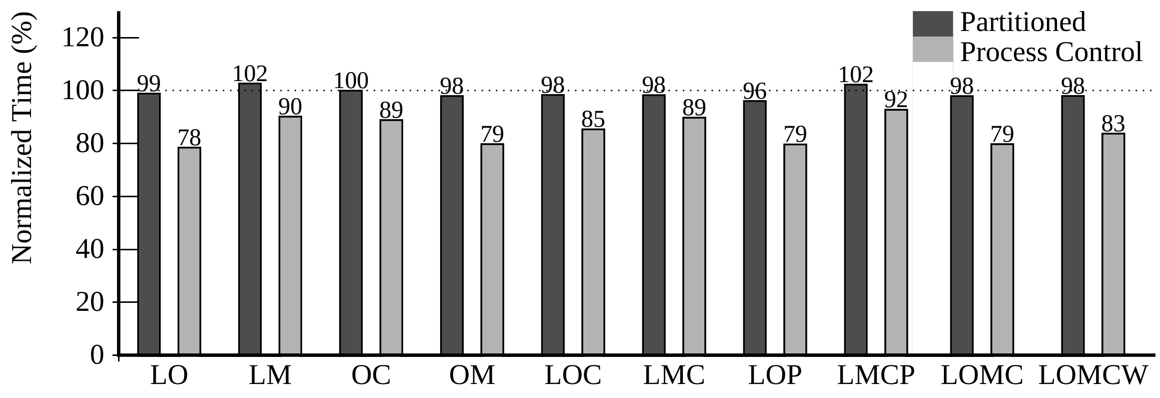

Figure 7.1: Performance of workloads under processor partitioning alone, and processor partitioning with process control, normalized with respect to standard UNIX performance.

and the gang scheduling results in Chapter 5, we see that process control is also superior to these. Figure 7.2 compares the geometric mean of all three policies across the suite of workloads. We see that process control attains a 16% performance advantage over standard UNIX scheduling, as compared to 6% with affinity scheduling and 3.5% with gang scheduling.

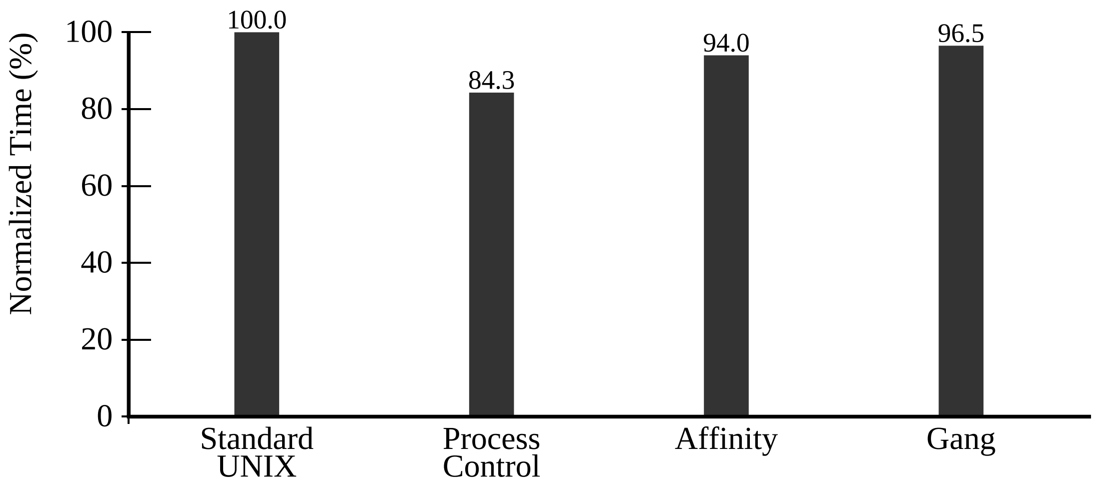

Figure 7.2: Geometric mean of normalized execution times of workloads using standard UNIX scheduling, process control, cache affinity scheduling, and gang scheduling.

There are two main reasons for this. First, with process control and processor partitioning, each application runs in its own environment with almost no interference from other applications. More importantly, as we can see by comparing the performance of the simply partitioned approach with the process control approach, the application processes do not interfere with each other. This lack of interference results in better cache behavior and leads to higher performance. Second, with process control, each application runs at a better "operating point" on its speedup curve.

The result is that in the presence of multiprogramming, techniques like cache affinity and gang scheduling exhibit low processor efficiency, since they use all the processors in the machine for each application. In contrast, the process control approach dynamically partitions the machine into several smaller machines, one per application, with each smaller machine providing higher processor utilization.[1]

It is interesting to compare the performance of the real process control implementation with the "ideal" model of running applications at their optimal operating point developed in Section 3.5. The ideal model resulted in an average performance gain of 23%. The gains of the real implementation are lower for a number of reasons. These include the overheads associated with process control (partitioning, scheduling, and signaling). Additionally, real applications cannot instantly change the number of processes they use as was assumed in the previous model. Finally, external events (such as network interrupts) can affect the performance of a workload on a real system. Given these problems, process control seems to do a good job of attaining much of the performance that is possible when problems (direct and indirect) associated with multiprogramming are eliminated.

The following subsections consider specific workloads in Figure 7.1, and additional workloads created to demonstrate particular points. The goal is to show the robustness of process control in a wide variety of contexts.

7.1.1 High Application Load

Although the process control strategy appears to work well with a small number of parallel applications, the question arises of how it performs when the number of applications exceeds the number of processors in the system. Workload LOMCW in Figure 7.1 shows the performance of process control, as compared with the standard UNIX system, when 5 parallel applications are run simultaneously on our 4-processor system. Those applications include `LocusRoute`, `Ocean`, `MP3D`, `Cholesky`, and `Water`.

With process control, the five parallel applications are each placed in their own processor sets. Although the kernel cannot allocate one processor to all applications simultaneously, it can allocate one processor to each of four applications, and reallocate processors so that all applications receive equal processing time. During this time, the applications all reduce the number of processes they use to one. Even if an application has no processors assigned to it, it cannot reduce beyond one process, since then it will have no way to resume suspended processes. Reducing beyond one process is also not necessary, since an application is effectively blocked when there is no processor allocated to run it. The overall result is that 5 processes, each from one application, time-share 4 processors. Unlike traditional time-sharing, however, the duration between process schedulings is the 300 ms reallocation interval. This contrasts with the normal 30 ms time quantum of the standard system. The result is that, although the performance is not

[1]As an interesting aside about the workings of the processor allocation module, observe that with process control and three or five parallel applications (as in workloads LOC, LMC, and LOMCW), the four processors in the SGI 4D/340 do not divide evenly among the applications. As discussed in Section 6, the policy module switches one of the processors between the applications periodically (every 300 ms) to maintain fairness. Since the overhead of the processor reallocation is small and it is done infrequently, the process control approach performs quite well.

as good as it would have been if there had been enough processors to prevent any time-sharing, the longer effective context-switch interval makes the performance much better than the standard UNIX system. The reduced number of competing processes also helps improve the performance of the process control approach.

7.1.2 Mixing Controlled and Non-controlled Applications

Most of the workloads in Figure 7.1 consist solely of process-controlled parallel applications. We now focus on the performance of the process control approach when applications that do not respond to process control are also run concurrently. Workloads LOP and LMCP combine compute-bound parallel applications that may use process control with a parallel make (`Pmake`), which cannot use process control (since it does not have a task-queue structure). When the workloads are run using process control, `Pmake` is run in the default processor set.

Figure 7.3 compares the performance of workloads LOP and LMCP under the standard UNIX scheduler and the process control approach in detail. The overall performance of each workload is broken into the components contributed by each application. The data shows that the process control approach performs significantly better than the standard UNIX scheduler for the process-controlled applications, and that the `Pmake` performance is similar with and without process control. According to the processor allocation policy described in Chapter 6, multiple processors are assigned to the default processor set if all 4 compiler processes are running. If the number of runnable processes drops due to I/O (as frequently occurs in `Pmake`), some of the processors are switched back to the other applications.

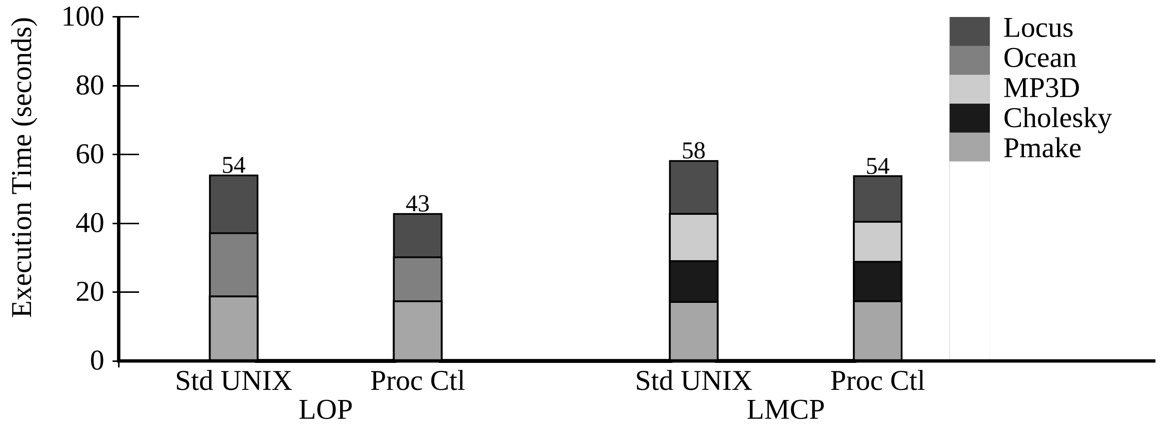

Figure 7.3: Performance of workloads LOP and LMCP under process control, normalized with respect to standard UNIX performance.

One of the effects of our processor allocation system is that the proportion of processor time received by a serial application is much higher than with a standard scheduling system. Under a UNIX scheduler, a single serial application running concurrently with a number of parallel applications receives a small portion of the processor time. If three parallel applications are being run at the same time as a serial application on a 4-processor system, as in workload LMCP, the serial application receives only up to about 1/13 of the total processor time, or less than

1/3 of a single processor. With process control, on the other hand, the processing time received by the serial application is on par with that of the parallel applications. This leads to greater fairness between applications regardless of the number of processes used and avoids incentives to increase processes in order to receive more processor time. Although the CPU time received by the parallel applications is reduced with our scheduler, as compared to the standard UNIX scheduler, the better efficiency of process control more than compensates. Other researchers have similarly found that allocating CPU time fairly to each job, regardless of type, is the best way to minimize the mean running time of all jobs in the system [26].

7.2 Response Time Issues for Interactive Applications

Another interesting class of serial applications is interactive programs, where the applications sleep for long periods of time broken by short periods of CPU use. Applications of this type abound on timesharing systems, from network servers to editors. The important factor in the performance of these applications is response time, not throughput. In our system we have sacrificed some response time in favor of parallel application throughput by allowing all processors to be taken away from the default processor set (see Section 6.3). We now examine how much effect this tradeoff has on response time. To test this aspect of performance, we implemented an application that repeatedly sleeps for a random period of time, then wakes up and executes for a short time before sleeping again. By comparing the measured duration of the sleep with the actual duration that was requested, we can tell how long it took to run the process after it woke up.

We ran this application simultaneously with workload LMC, which consists of the `LocusRoute`, `MP3D`, and `Cholesky` applications. We compared the performance under the standard UNIX scheduler and under process control. The parallel execution times and approximate response times are shown in Figure 7.4.

With the standard scheduler, the average response time is fast (5 ms) but the performance of the parallel applications, as seen before, is poor (average execution time for the applications is 72 seconds). With process control, the response time is not quite as good (21 ms) but the performance is much better (average execution time is 54 seconds). The response time is worse for process control because all processors are assigned to non-default processor sets most of the time. When the application in the default processor set becomes runnable, there is a time lag before a processor is reallocated to the default processor set. In addition there may be extra delay while higher-priority processes (also in the default processor set) are executed. We believe that the added response time is small enough to be negligible.

7.3 Performance in the Presence of I/O

As stated in Section 6.2, when a process blocks on I/O, a signal is sent to the application so that it may create or resume a process to run on the processor just made idle. So far, however, our experiments have not shown the benefits of this feature since our applications perform very little I/O. To generate more I/O activity in the system without changing the applications, we locked

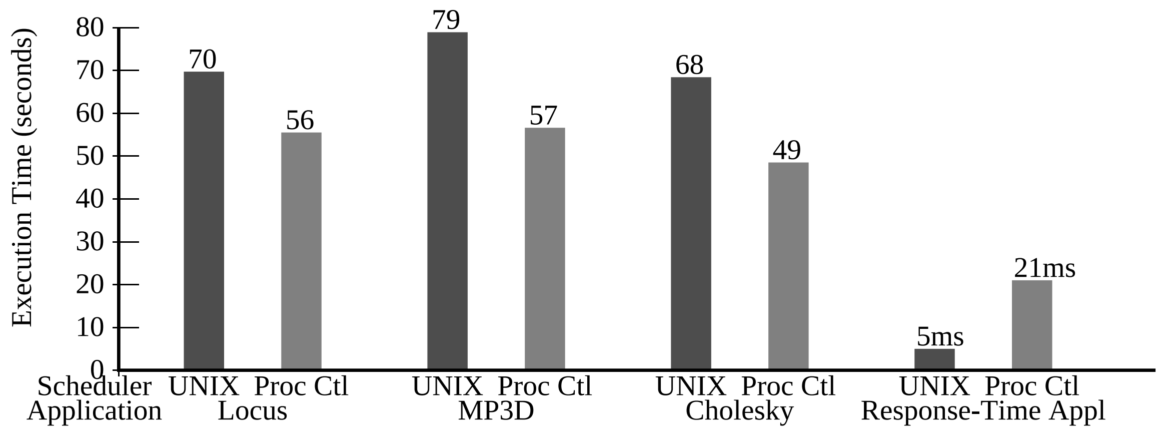

Figure 7.4: Performance of workload LMC running simultaneously with an interactive serial application, along with system response time. Performance is shown both for the standard UNIX scheduler and the process control scheduler.

a number of pages in physical memory, effectively reducing the amount of memory available to other applications. We used a machine with 32 Mbytes of physical memory, and locked in between 4 and 16 Mbytes of memory. We then ran `Cholesky` using process control. We compared two variants of I/O signaling. In one case, we turned off the signals that are generated when a process blocks on I/O. In the other case, these signals are turned on. The execution times as the amount of memory taken away is varied are shown in Figure 7.5. Regardless of the amount of memory reserved, performance is consistently best with I/O generated signals turned on. With 4MB and 8MB taken away, the execution times of both approaches are essentially equal. There is still enough memory in the system to avoid paging. With 12MB taken away, `Cholesky` took 35 seconds with signals turned on, and 30 seconds with I/O signals turned off. With 16MB taken away, the `Cholesky` execution thrashes, taking 175 seconds with signals turned on, and 125 seconds with signals turned off. Although the use of I/O signals does not completely avoid the problem of poor memory system behavior when the working set of a program does not fit into physical memory, it does help avoid aggravating the problem with excessive processor idle time waiting for pages.

7.4 Performance under Varying System Load

Until now, all of the experiments done with the workloads have involved synchronizing their constituent applications before each enters its parallel section, and timing only the parallel code. This measures the performance of scheduling policies under a steady load of parallel applications, but does not model well the case of varying loads, where applications are frequently entering and leaving the system. We will now look at the performance, including both parallel and serial time, of the workloads when run without inter-application synchronization and with the applications started at 10 second intervals.

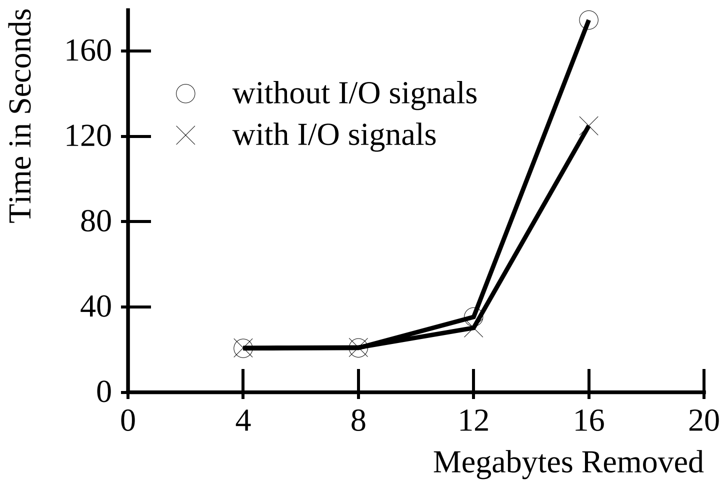

Figure 7.5: Performance of `Cholesky` application with pages reserved. Data is presented with I/O generated signals turned on and turned off.

Figure 7.6 shows the performance of the LOC workload under standard UNIX scheduling and process control, respectively. The workload consists of `LocusRoute`, `Cholesky`, and `Ocean`, with applications staggered at 10 second intervals, and no synchronization of parallel sections. The figure shows the number of processes running at each point in time. In the standard UNIX case, there are as many as 12 processes running in the system when all three applications are running in parallel simultaneously. In the process control case, the process control system serves to limit the number of processes to near 4 at all times. The process control execution completes in 49 seconds, as compared to 56 seconds for the standard UNIX scheduling system, an improvement of 15%. This is the same as the improvement for process control when the applications are synchronized and only parallel execution is measured. The performance of most of the individual applications is also improved under process control; `LocusRoute` takes essentially the same time in each case, but `Cholesky` takes 35 seconds with process control rather than 40 (an improvement of 15%) and `Ocean` takes 28 seconds rather than 36 (an improvement of 29%).

7.5 Conclusions

In conclusion, we see that the process control approach performed much better than the standard UNIX scheduler for all of our workloads. In addition, it performed substantially better than either cache affinity or gang scheduling. The performance improvements held even when the number of parallel applications exceeded the number of processors in the system. The process control approach was also shown to work well when non-process-controlled and interactive applications were run concurrently with process-controlled applications; processors were allocated in such

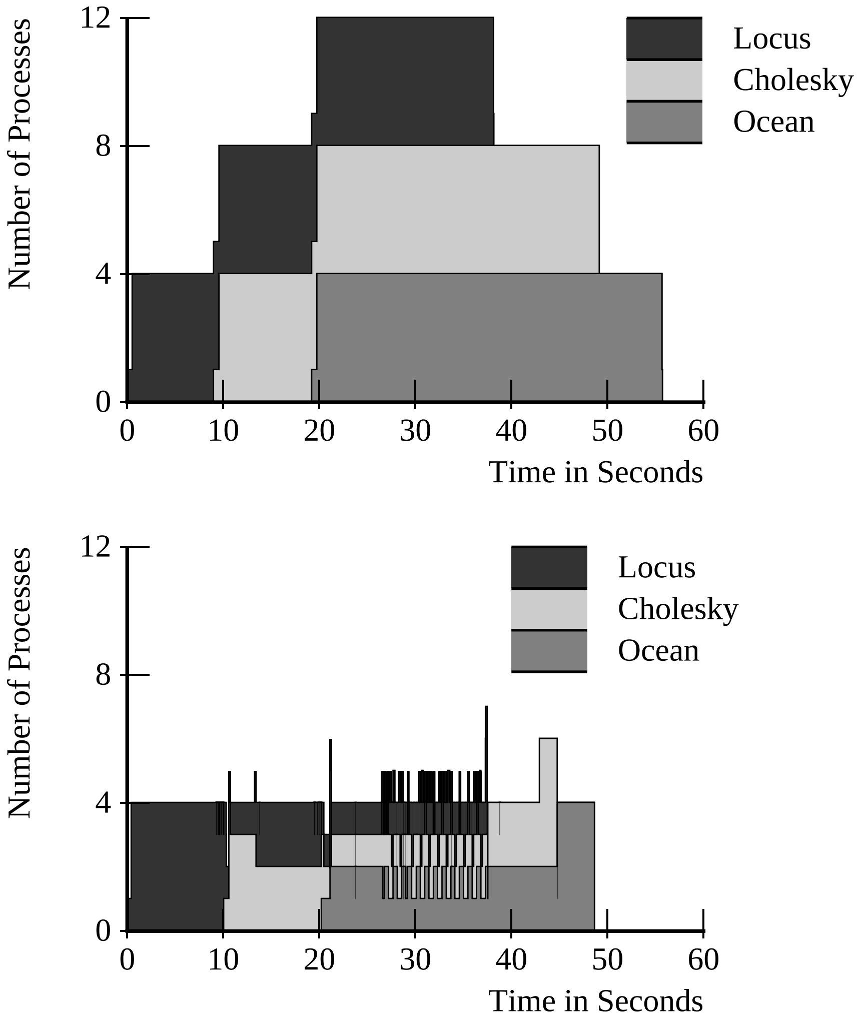

Figure 7.6: Performance of workload LOC with applications started at 10 second intervals, under standard UNIX scheduling and process control.

a way as to preserve the performance of process-controlled applications while retaining fast turnaround and response time for other types of applications. We also studied the performance of the process control approach in the presence of a significant amount of I/O. The use of kernel notifications when processes block on I/O kept processors from idling, though the difference in overall performance was small until the I/O activity became substantial. Finally, process control also performed well under conditions of varying system load, where applications enter and leave the system at staggered intervals.

Chapter 8

Conclusions

8.1 Contributions

This thesis has investigated several approaches to efficiently scheduling multiprogrammed applications on a shared-memory multiprocessor. These approaches varied in complexity and in the extent of modification to existing scheduling systems, from selectively boosting the priority of appropriate process-processor pairs (cache affinity scheduling), to boosting the priority of entire applications simultaneously (gang scheduling), to primarily allocating processors to applications spatially rather than temporally (process control). This thesis thoroughly evaluated and compared these scheduling approaches in a consistent manner by measuring the performance of each approach running a fixed set of realistic workloads. Implementations of the approaches were developed using the same base operating system and the workloads were run on the same machine.

The first two approaches have been proposed by others. The thesis work developed new techniques for implementing each approach simply and efficiently, and studied the effect of each on a suite of parallel and serial workloads. Both cache affinity scheduling and gang scheduling attained small performance gains under most workloads. The limited extent of the performance gains appears to be the result of inherent factors of the applications and workloads, not problems with the scheduling implementations.

The thesis then proposed an approach to multiprocessor scheduling that uses cooperation between the kernel and applications to execute applications efficiently. The approach, called *process control*, involves providing support for applications to dynamically match the number of processes they use to the number of processors that are available to them. Processors are partitioned among applications in such a way that multiple applications may be run simultaneously while preserving fairness. The partitioning also serves to promote stable running environments and improves cache efficiency.

An important facet of the structure of the system is that it allows flexibility in the applications' use of process control. The decision of when a process should be suspended, resumed, or created, is left to user-level software, rather than being embedded in the kernel. This also allows our modifications to fit into the standard UNIX model of process scheduling; the system still supports run queues, time slices, process priorities, and other standard UNIX mechanisms. In fact, if there is only one active processor set in the system, processes are scheduled exactly as they would be in a standard UNIX system.

The process control approach performed much better than the standard UNIX scheduler when the number of processes exceeded the number of processors under both single-application and multi-application conditions. The better performance can be attributed to two main factors. First, process control offered significantly reduced context switch rates, eliminating direct context switch overhead and indirectly providing better cache hit rates and synchronization performance. The hit rates improved since there were no intervening processes to displace useful data from the cache. Second, a more subtle factor, the performance also improved because the process control approach allowed each application to run at a better "operating point" on its speedup versus processors curve. By running each application with fewer processes, process control helped improve cache hit rate (reducing false-sharing and communication misses), load balancing, and synchronization overhead for the applications. Our tests in Chapter 3 indicated that the operating point effect is the main source of performance gains for applications using process control.

The process control approach also performed better than the alternative approaches of cache affinity scheduling and gang scheduling. Process control provides a more comprehensive answer to the problems of synchronization performance due to multiprogramming and cache data loss, virtually eliminating both among applications able to fully take advantage of process control. Process control is also the only approach that takes advantage of the operating point effect.

8.2 Extensions and Future Work

The results in this thesis have been based on a machine with only four processors. An obvious question that arises is how the results would scale to larger machines. The next subsection discusses the effect of using a single-bus machine with more processors. The following subsection discusses issues involved in scheduling on scalable multiprocessors.

8.2.1 Larger UMA Machines

An increase in the number of processors in a machine, without changes to the rest of the machine structure, would have little effect on the problems of cache data replacement and synchronization issues involved with multiprogramming. Process migration may be slightly increased in a standard UNIX scheduler, since there will be even less probability that a process will randomly be selected to run on the processor it previously ran. Synchronization problems may be increased a bit more substantially since applications will be run with more processes, necessitating more synchronization. However, by far the largest difference will be in the operating point effect. The efficiency of an application not only tends to decrease as more processes are used, but the rate of decrease *increases* with more processes. For example, if an application runs 20% more efficiently with two processes than with four, it might run 40% more efficiently with four processes than with eight. The result is that the operating point effect becomes even more significant in application performance.

Thus, process control should perform even better on a machine with more processors. I have performed some experiments on a SGI machine similar to the one used for the results in this thesis, but with eight processors. The results showed an improvement from process control that is substantially greater than the results presented in Chapter 7.

8.2.2 NUMA Machines

This thesis has been devoted to studying problems that arise when running multiprogrammed workloads on single-bus shared-memory machines, with small to moderate numbers of processors. As system designers try to build machines with more processors, they run into an inherent limitation in the single-bus design, as the bus begins to saturate with traffic. The alternative is to build a scalable system, using a point-to-point network connecting clusters of one or more processors on a bus. Shared-memory mechanisms can still be supported in hardware. The Stanford DASH machine [25] is such a machine, with 4-processor clusters. [1]

On many scalable machines, remote memory accesses (accessing memory in another cluster) take much longer than local memory accesses (accessing memory in the same cluster). [2] These are also called NUMA (Non-Uniform Memory Access) machines because of this effect. The result is to increase the cost of inefficiencies from a naive approach to scheduling multiprogramming workloads. Process migration between clusters, and scheduling applications on more than one cluster, can have a substantial cost. The scheduling solutions to this problem must also be made more sophisticated. For example, cache affinity scheduling must distinguish between a process's affinity for a processor and its affinity for a cluster (or memory). The latter may actually be more important than the former. The policy module of a process control system should likewise pay attention to cluster boundaries (in machines where clusters consist of more than one processor) and partition processors as much as possible along those boundaries. System-wide gang scheduling may be unreasonable when the cost of migrating data between clusters is high; however, a hierarchical gang scheduler (as suggested by Feitelson and Rudolph [14]) where applications are gang-scheduled within clusters or small groups of clusters might work well. In sum, the advent of scalable shared-memory machines creates a great opportunity for improving the performance of multiprogrammed workloads with intelligent scheduling policies. Several researchers have begun to work on these problems in the context of NUMA machines [9, 14, 34, 49].

8.3 Final Remarks

In conclusion, process control has been shown to be a simple but effective approach to improving the performance of parallel applications on multiprogrammed systems, one that may be cleanly integrated into a traditional operating system. Although other scheduling policies, like cache affinity and gang scheduling, may provide some performance benefits as compared to a standard scheduler, process control provides more substantial benefits to a wider class of applications.

As applications and user-level support packages become more and more sophisticated, it

[1]As mentioned in Chapter 2, the machine used in the experiments for this thesis was actually a single-cluster DASH machine. The DASH machine was used in preference to a standard SGI machine because of the additional performance measurement hardware available.

[2]Note that some machines have only one processor per cluster, associating local memory with each processor.

becomes increasingly important to inform them about kernel-level events. Process control is only one example of the use that can be made of information about resource management. Others have considered informing applications of the state of the virtual memory system [20]; an application might try to reduce its memory usage in an attempt to keep its working set in physical memory. In a NUMA machine that allowing process migration, it can be important to give applications information on memory and process placement. Database programmers, trying to tune their applications to run at optimal performance, often desire information on scheduling and memory management. While operating systems are often designed with the intention of hiding as many of the low-level details as possible from the user, this work indicates that future operating systems designers should consider revealing some of these details to applications that can make good use of the information.

Appendix A

Process Control Interface

In Chapter 6 we saw the necessary characteristics of an application to use process control. This appendix describes the interfaces that applications (or threads packages, parallel languages, etc.) with these characteristics can use to implement process control on the Silicon Graphics system. Section A.1 describes system calls added to IRIX to put an application in its own processor set and cause signals to be sent to the application telling it the number of processes it should be using. Section A.2 describes calls to a library that provides a simpler user-level interface to applications, easing the transition to using process control.

A.1 Kernel Interface

The interface to the kernel for process control is implemented using the prctl() call:

```
    int prctl(flag, var1, [var2])
    int flag, var1, var2;
```

The process control commands are represented by specific flags.

```
    prctl(PR_PSETMAT, num)
```

`PR_PSETMAT` creates a separate processor set for the current process (or group of processes sharing an address space) and tells that the application will initially create `num` processes to run in this processor set.

```
    prctl(PR_PSETNEED, type, val)
```

`PR_PSETNEED` informs the kernel of changes in the kernel setting of the maximum number of processors the current application (process or group of processes sharing an address space) can use. `type` is either `NEED_REL` or `NEED_ABS`, and denotes whether `val` is to be treated as a relative change to the current setting, or an absolute value to replace the current setting, respectively. A `type` of `NEED_ABS` and a `val` of `-1` informs the kernel that the application can use an unlimited number of processors; this is the default for a new processor set.

`prctl(PR_PSETASS, pid, psetid)`

`PR_PSETADD` assigns process `pid` (or the group of processes sharing an address space that includes process `pid`) to processor set with id `psetid`. This can be used to run multiple applications within a single processor set (other than the default).

`prctl(PR_RMPSET, psetid)`

`PR_RMPSET` deletes the processor set with id `psetid` and moves all processes and processors currently assigned to that processor set to the default processor set.

`prctl(PR_GETPSET)`

`PR_GETPSET` returns the processor set id of the current process. The default processor set has an id of 0.

`prctl(PR_STRTPSET)`

`PR_STRTPSET` signals to the kernel that all setup operations for process control have been performed by the application associated with the current process, and process control signals can now be freely sent (including those that have been stored up pending this call).

`prctl(PR_SETDPCTRL)`

`PR_SETDPCTRL` tells the kernel to allocate the shared counters used to indicate the number of processes that the current application should use, and prepare to send process control signals.

`prctl(PR_GETDPCTRL)`

`PR_GETDPCTRL` returns a value indicating whether or not the shared counters and process control signals have been initialized (by `PR_SETDPCTRL`) for the current application.

`prctl(PR_BLOCKIGN)`

`PR_BLOCKIGN` tells the kernel not to send explicit signals to the current application in reaction to an event that indicates an application process should block; for example, a processor being taken away from the application, or a process returning from an I/O operation. This is used to optimize performance as described in Chapter 6.

Any operations associated with process control not described here are performed automatically by other kernel operations. For example, if a new process is created that shares the same address space as the parent process, the child process is placed in the same processor set as the parent. If the new process has its own address space, it is assumed to belong to a different application and is placed in the default processor set. When the last process in a processor set (other than the default) exits, the processor set is automatically deleted.

A.2 Library Interface

Setting up the user-level support for process control can be moderately complicated. I have designed a library to ease the problem for implementors of parallel languages and threads systems by providing a much simpler interface than kernel calls listed above. The language/thread writer simply needs to call the library when appropriate, and design a system for blocking and unblocking processes when notified. The library implements a process control system with one client application per processor set. The library calls (with descriptions of when they should be called) are as follows.

```
void pctrl_init(maxprocs, currprocs, signals, handler)
int maxprocs, currprocs, signals;
void (*handler)();
```

`pctrl_init` initializes the process control system, creating a processor set for the application and setting up the shared counter and signal handler. `maxprocs` is the maximum number of processes the application wishes to use (a value of `-1` denotes an unlimited number). `currprocs` is the number of processes that will automatically be created by the application. This call should be made *before* processes other than the master process have been created. `signals` is a boolean value denoting whether or not the process control signals should be used. `handler` is the function that should be called when the number of processes should be changed; it will be passed a number indicating the number of processes to be created/resumed (if positive) or blocked (if negative). `handler` will be called either in response to a kernel-generated signal or due to an application-originated `pctrl_check` call.

```
void pctrl_check()
```

`pctrl_check` checks the value of the shared counter and calls the application-defined handler if appropriate. This should be called whenever the application is in a safe suspension point; e.g., when it has finished executing one task or thread and is about to select another to execute.

Bibliography

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 95–109, October 1991.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.

[3] Forest Baskett, Tom Jermoluk, and Doug Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. In *Proceedings of COMPCON '88*, pages 468–471, February 1988.

[4] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[5] James Boyle, Ewing Lusk, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, 1987.

[6] Rohit Chandra, Anoop Gupta, and John Hennessy. *COOL: A Language for Parallel Programming*. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1990.

[7] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, 1988.

[8] Mark Crovella. The costs and benefits of coscheduling. Technical report, University of Rochester Computer Science Department, May 1991.

[9] Mark Crovella, Prakash Das, Cezary Dubnicki, Tom LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Computing*, pages 590–597, December 1991.

[10] Murthy Devarakonda and Arup Mukherjee. Issues in implementation of cache-affinity scheduling. In *Proceedings Winter 1992 USENIX Conference*, pages 345–357, January 1992.

[11] Thomas W. Doeppner, Jr. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Brown University, 1987.

[12] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, March 1989.

[13] Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. In *Proceedings of the USENIX Workshop on UNIX and Supercomputers*, pages 1–17, 1988.

[14] Dror G. Feitelson and Larry Rudolph. Distributed hierarchical control for parallel processing. *IEEE Computer*, 23(5):65–77, May 1990.

[15] Stuart I. Feldman. Make: A program for maintaining computer programs. Technical Report 57, Computing Science, Bell Laboratories, 1977.

[16] Richard P. Gabriel and John McCarthy. Queue-based multi-processing Lisp. In *ACM Symposium on Lisp and Functional Programming*, pages 25–43, 1984.

[17] Anoop Gupta, Charles Forgy, Dirk Kalp, Allen Newell, and Milind Tambe. Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2), 1988.

[18] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of SIGMETRICS '91*, pages 120–132, May 1991.

[19] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[20] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, September 1992.

[21] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for shared-memory multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 41–55, October 1991.

[22] Monica S. Lam and Martin Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[23] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[24] Dan Lenoski, Kourosh Gharachorloo, James Laudon, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of scalable shared-memory multiprocessors: The DASH approach. In *Proceedings of COMPCON '90*, pages 62–67, February 1990.

[25] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[26] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of SIGMETRICS '90*, pages 226–236, 1990.

[27] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of SIGMETRICS '88*, pages 104–113, 1988.

[28] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors. Technical Report 90-03-02, Department of Computer Science, University of Washington, February 1991.

[29] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[30] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, April 1991.

[31] John K. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[32] Jonathan Rose. LocusRoute: A parallel global router for standard cells. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, June 1988.

[33] Edward Rothberg and Anoop Gupta. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, November 1990.

[34] Sanjeev K. Setia, Mark S. Squillante, and Satish K. Tripathi. Processor scheduling on multiprogrammed, distributed memory multiprocessors. In *Proceedings of SIGMETRICS '93*, pages 158–170, May 1993.

[35] Silicon Graphics Inc. *IRIX 4.0 Programmer's Reference Manual*, August 1991.

[36] Jaswinder Pal Singh and John L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

[37] Jaswinder Pal Singh and John L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experience, results, and implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[38] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[39] Larry Soule and Anoop Gupta. Characterization of parallelism and deadlocks in distributed digital logic simulation. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, June 1989.

[40] M. Squillante and R. Nelson. Analysis of task migration in shared-memory multiprocessor scheduling. In *Proceedings of SIGMETRICS '91*, pages 143–155, May 1991.

[41] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[42] Shreekant S. Thakkar and Mark Sweiger. Performance of an OLTP application on Symmetry multiprocessor system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 228–238, May 1990.

[43] Dominique Thiebaut and Harold S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[44] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the benefits of cache-affinity scheduling in shared-memory multiprocessors. Technical Report CSL-TR-92-536, Computer Systems Laboratory, Stanford University, August 1992. Published in short form in the *Proceedings of SIGMETRICS '93*, pages 272–274, May 1993.

[45] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, 1989.

[46] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 26–40, October 1991.

[47] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Seminar on Performance of Distributed and Parallel Systems*, pages 455–472, December 1988.

[48] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of SIGMETRICS '90*, pages 214–225, 1990.

[49] Songnian Zhou and Timothy Brecht. Processor pool-based scheduling for large-scale NUMA multiprocessors. In *Proceedings of SIGMETRICS '91*, pages 133–142, May 1991.